TUM

INSTITUT FÜR INFORMATIK

RTSE'97 – Workshop on "Requirements Targeting Software and Systems Engineering"

Manfred Broy
Bernhard Rumpe

TUM-I9807
April 98

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM-INFO-04-I9807-50/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final Report

4. TITLE AND SUBTITLE

Workshop on "Requirements Targeting Software and Systems Engineering"

5. FUNDING NUMBERS

DAAG55-97-1-0289

6. AUTHOR(S)

Manfred Bray, principal investigator

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES)

Institute for Informatik
Muncheu, Germany

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

ARO 37363.1-MA-CF

11. SUPPLEMENTARY NOTES

The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12 b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

no abstract funished

14. SUBJECT TERMS			15. NUMBER IF PAGES
			16. PRICE CODE
17. SECURITY CLASSIFICATION OR REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

Foreword

In Europe and North America the competence in software engineering research has different profiles. While in North America there is a lot of know how in the practical, technical, and organisational aspects of software engineering, in Europe the work concentrates more on foundations and formal modelling of software engineering issues. Both approaches have different strengths and weaknesses. Solely practice driven research in software engineering is in the danger of developing into a shallow field and could fail to find a solid scientific basis and contribute substantially to the progress in software engineering. Work concentrated on formal aspects only is in the danger of becoming too theoretical and isolated from practice such that any transfer into practical application will fail.

Substantial progress in software engineering can be achieved by bringing together pragmatic and foundational work in software engineering research. This can provide a step towards a more common scientific basis for software engineering that allows us to integrate the various results of research and workshops, leading to fruitful synergetic effects. It will also help to identify critical research paths and developing an adequate paradigm for the scientific discipline of software engineering.

It was the goal of this workshop to bring together experts from science and practice in software and systems engineering from North America and Europe.

In software and systems engineering it is necessary to distinguish the enormous difference between the dynamics in development we refer to and the limited scope assumed by many of today's software managers that still use outdated techniques. Many of the unsolved problems associated with the old techniques are symptoms of lack of formalization and lack of automation support.

The intended focus of the workshop was on unified sets of formal models and associated methods suitable for automation for many aspects of software development, in particular those that address change and those that apply on a large scale. Some of the intended aspects of software evolution are

- modifiable software architectures,
- resource changes,
- context changes,
- requirements changes,
- changes to decomposition structures, and
- changes in plans.

These issues are related to formal representations of the version history, and formal representations of the activities that produced existing versions or have been proposed to produce future versions.

The essence of the problem is to establish and maintain consistency among various kinds of software artefacts throughout the development and evolution process, including consistency between requirements, architectures, and programs. Automation support is needed to determine dependencies and to use this dependency information to provide decision aid for software synthesis, analysis, and evolution. Many versions of each artefact are produced as the software evolves, and changes to the dependency structure must be recognised and reacted to. The challenge is to better formalise the problems in this area, and to develop some of the badly needed technical solutions.

If we as a community can succeed in this, the results will provide convincing evidence that formal methods can have strong practical value, and help reverse the trend of weakening support for the subject from both industry and governments. It seems that previous work on formal methods can be applied to problems related to these topics, but it may require non-traditional approaches. The challenge helped to trigger new ideas at the workshop, and perhaps opened new opportunities for progress.

It is well recognised in the meanwhile that software and systems engineering as an important issue in technical systems still lack a proper scientific basis. The many efforts in academia, especially under the heading formal methods, towards such a scientific basis have produced many valuable and interesting scientific results; however, most of the work of integrating this with the practice of software engineering is still missing. Nevertheless, we can observe a starting trend to bring together practical considerations and approaches with scientific results. A good example is the Unified Modelling Language that recently was designed and still will evolve. The fact that a proper semantic basis is needed for a proper methodological support is much more recognised than in its predecessors. Nevertheless, more efforts are necessary to give the scientific research more focus w.r.t. the questions that are important for practice and to stimulate a transfer between academia and application. It was the goal of the workshop to contribute to this task.

The workshop took place in early October 1997 in Bernried in Germany. It fulfilled the expectations formulated above. It is our pleasure to thank Sascha Molterer for his excellent help in organising the workshop and the Army Research Office and in particular Dave Hislop for the generous financial support.

March 1998

Manfred Broy, Bernhard Rumpe

Table of Contents

A Discipline for Handling Feature Interaction

Egidio Astesiano – Gianna Reggio
DISI
Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova
Via Dodecaneso, 35 – Genova 16146 – Italy
{ astes,reggio } @ disi.unige.it
http://www.disi.unige.it

1 Introduction

Evolution in software development has many facets. One which emerged in the last five years, especially in the area of telecommunications and networking, is the continual expansion of services. Recognizing that objected-oriented incrementality is not adequate to cope with this new problem in full generality, the concept of "feature" as unit of update has been introduced, see e.g., [10], and taken as a pivotal unit for even new paradigms, like feature-oriented programming, feature-oriented specification and so on [7].

In spite of the considerable effort (see some pointers to recent work at the end), still many issues deserve further attention and investigation, as it is admitted by the specialists of the subject, also taking into account the growing complexity of the applications concerned. Among the issues, feature composition and interaction is definitely the one attracting most attention. This is also witnessed by the success of an International Workshop on Feature Interaction, now reaching in '98 its fifth edition. In particular a lot of work is reported on the so-called feature interaction detection, possibly done automatically. According to this viewpoint, feature interaction is synonym with unexpected/ unwanted results.

We are among those sharing the view that the problem of feature-interaction should be tackled within a wider methodological approach. This view is best expressed by Pamela Zave in [10], who calls for "an approach based on modular specifications and separation of concerns ...(aimed) to organize the specification so that it is easy to add without destroying its structure or desiderable properties". This is indeed the underlying challenging problem; again in P. Zave's words "the goal of extendible specifications is as difficult to achieve as it is easy to state". For example, in the realm of reactive and concurrent systems, the classical approach to incrementality has been based on the notion of process/ agent as unit of change; feature-driven incrementality deals instead with incrementality within a process and refers/affects the behaviour of the pre-existing processes.

In this paper we want to outline a specification framework supporting a feature-driven software development method with rigorous semantics, offering conceptual tools for expressing requirements on unwanted interactions. The framework intends to be adaptable

to a variety of different application fields, like telecommunications, information systems; clearly, depending on the application, sensible domain specific methods should be derived.

The specification framework we present is built over a general specification formalism for reactive and concurrent systems (labelled transition logic [1]), already supporting componentwise modularity. Its adaptation to feature-driven modularity is based on the key principle of supporting the separation of concerns by factorizing the specifications. Indeed we basically provide various levels of control on interactions:

- with every feature some interaction requirements are associated, intended to provide constraints on the outcome of its composition with other features;
- for the composition, a flexible concept of "compatibility" is introduced, factorized in turn w.r.t. the different components of feature specifications; thus a choice is possible between different compatibility criteria;
- then composition is allowed only for compatible pairs and follows a general schema.

The developer is guided to analyse the outcome of the composition by looking at different possible kinds of interactions, whose "goodness" or "badness" may depend on the application.

Let us now outline from a more detailed technical viewpoint, the main features and perhaps novelties of our proposal.

A system is modelled by a labelled transition system (Sect. 2.1), whose states are sets of attribute values (as in many O-O approaches); the transitions denote action capabilities, with the labels indicating the exchanges with the external environment; transitions can be grouped under action names, essentially indicating the kind of event to which the transition refers, for the purpose of feature composition.

In this paper we restrict ourselves to consider simple systems, i.e. we disregard the fact that a system may be composed by other subsystems; this is not a restriction from the methodological point of view, but the introduction of component modularity poses further interesting technical problems, which are the subject of some still ongoing investigation.

We address the specifications at the design level, namely for characterizing essentially one system; this is achieved by associating a labelled transition system with a specification. A feature specification is basically the specification of a labelled transition system with some interaction requirements, i.e. formulae constraining the result of adding other features.

Together with a *basic semantics* (the associated labelled transition system), we introduce the novel concept of *complete semantics*, which consists of all the specification models satisfying the interaction requirements, under a kind of anti-frame assumption: an attribute value may be changed unless its invariance is explicitly stated. The concept of complete semantics plays a major role in composing features and understanding their interaction. In order to reason about complete semantics (which usually admits infinitely many models), we propose a kind of canonical labelled transition system representation (an abstract interpretation), which allows to reason concretely about systems including other features.

Both concepts, basic and complete semantics, apply in turn to the result of feature composition, which is again a feature. The analysis of feature interaction is done against

the original features at the basis of the result of their composition; this comparison can be done w.r.t. different interaction criteria, some of which refer to a comparison of the overall behaviour, provided by a notion of feature simulation, reminiscent, but different from the classical (bi-)simulation of CCS and process algebras.

In this paper we try to illustrate the concepts by examples, introducing formalities only when required. Necessarily we discuss toy-examples, but an extended specification case study around the well-known telephone systems has been specified and analysed with our technique.

2 Feature Specifications and Their Semantics

2.1 Simple Reactive Systems

We distinguish reactive systems in *simple* and *structured or concurrent*; the latter are those having cooperating components, which are in turn reactive systems (simple or structured).

To model reactive systems we use labelled transition systems (see [6]).

A *labelled transition system* (shortly *lts*) is a triple

$$(STATE, LABEL, \rightarrow)$$

where $STATE$ and $LABEL$ are two sets, the *states* and the *labels* of the system, and $\rightarrow \subseteq STATE \times LABEL \times STATE$ is the *transition relation*. A triple $(s, l, s') \in \rightarrow$ is said a *transition* and is usually written $s \xrightarrow{l} s'$.

A reactive system R is thus modelled by an lts $LTS = (STATE, LABEL, \rightarrow)$ and an initial state $s_0 \in STATE$; the states reachable from s_0 represent the intermediate (interesting) situations of the life of R and the arcs between them the possibilities of R of passing from a state to another one. It is important to note that here an arc (a transition) $s \xrightarrow{l} s'$ has the following meaning: R in the state s has the *capability* of passing into the state s' by performing a transition, where label l represents the interaction with the external (to R) world during such move; thus l contains information on the conditions on the external world for the capability to become effective, and on the transformation of such world induced by the execution of the action; so transitions correspond to *action capabilities*. Later on we will see the use of labels, which, as in CCS ([6]), allows to represent open systems and their composition to build concurrent systems.

Here we assume that the states of simple systems are modelled by heterogeneous tuples with named components (records); accordingly to an O-O terminology they are determined by a set of attributes. Furthermore the labels are described by "constructors" possibly parameterized corresponding to the various *kinds* of interaction with the external world of the system. We do not give the precise syntax of our specifications, but just present an example.

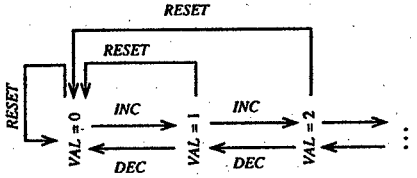
Example 2.1 Simple system specification As an example, consider the specification of a simple counter whose value may be incremented, decremented and reset to 0.

simple system COUNTER_S[c] =
data NAT
attrs VAL: *nat*
interface $INC, DEC, RESET()$
activity I: $c \xrightarrow{INC} c[VAL+1/VAL]$
if $VAL > 0$ **then** D: $c \xrightarrow{DEC} c[VAL-1/VAL]$
R: $c \xrightarrow{RESET} c[0/VAL]$

- NAT is a specification of natural values with a sort *nat* given elsewhere. This system has just one attribute VAL of type *nat* and three label constructors INC, DEC, $RESET$, in this case they are not parameterized, corresponding to receive the three corresponding commands. We use the keyword **interface** for the specification part introducing the labels, because in our setting labels are really the interface of the specified reactive system. The signature of NAT, the list of the attributes with their types and of the label constructors with the types of their arguments give the "signature" of the specification.
- c is the generic name for a state of the system.
- The activity of the system (labelled transitions) is specified by the conditional rules in the **activity** part: the system has a transition iff such transition is obtained by instantiating a rule in a way that the premises hold. In any state COUNTER_S may perform a transition labelled by INC incrementing by 1 the value of the attribute VAL and can be reset putting VAL to 0; while it may perform a transition labelled by DEC only if the value of VAL is bigger than 0 and in such cases VAL will be decremented by 1. $\_[\_/\_]$ as usual denotes the update operation.

  I, D and R denote the "actions" to which the transitions belong; their role will be explained in Sect. 3.1.

The semantics of COUNTER_S is essentially (because it includes also the used data structures, just a many-sorted first-order structure) given by the lts graphically depicted below. In this paper for simplicity we call lts also these richer structures.



The form of the given specification is a friendly, but rigorous, notation in an O-O style; it can be expanded, in a canonical way, into the following 1st order many-sorted specification with positive conditional axioms (Horn clauses), then the associated lts is the one obtained by logical deduction (this is what is called "initial" model in the algebraic community); see [9].

COUNTER_S =
use NAT
sorts *state, label, act*
opns $\mathsf{I}, \mathsf{D}, \mathsf{R}: \rightarrow act$
opns $\langle\_\rangle: nat \rightarrow state$
$VAL: state \rightarrow nat$
$\_[\_/VAL]: state\ nat \rightarrow state$
$INC, DEC, RESET: \rightarrow label$
axioms $VAL(\langle n \rangle) = n$
$\langle n \rangle[n'/VAL] = \langle n' \rangle$
$n = VAL(c) + 1 \supset \mathsf{I}: c \xrightarrow{INC} c[n/VAL]$
$VAL(c) > 0 \wedge n = VAL(c) - 1 \supset \mathsf{D}: c \xrightarrow{DEC} c[n/VAL]$
$n = 0 \supset \mathsf{R}: c \xrightarrow{RESET} c[n/VAL]$

End example

2.2 Feature Specifications

Intuitively a feature specification for simple reactive systems is a partial description of a simple system, intended as a description of only some "parts" of that system; in our setting these parts may be some basic data, part of the states (some attributes), part of the interface (some label constructors) and part of the activity (some action capability descriptions or partial descriptions of some action capabilities).

Technically this partial description may be just a simple system specification as introduced in Sect. 2.1. But, in our approach, a specification of a feature has another strongly relevant component, namely the *interaction requirements*, a description (specification) of the allowed interactions with other features.

Example 2.2 Counter as a Feature We specify a feature for simple systems, COUNTER, corresponding to have a counter with some commands for modifying it, just by adding to the system specification COUNTER_S of Ex. 2.1 an interaction requirement part, consisting of:

if $a: c \xrightarrow{l} c'$ **and** $VAL \neq VAL'$ **then** $(l = INC$ **or** $l = DEC$ **or** $l = RESET)$.

Here and in the following we shorten $VAL(c')$ to VAL'. The above axiom requires that any added feature cannot modify the attribute VAL as a result of a transition with labels different from INC, DEC or $RESET$. Thus other features may, e.g., add new attributes and extend transitions labelled with INC or DEC or $RESET$ to act on them, while it is not possible to add transitions modifying VAL with new labels.

Below we give an example of a simple system incorporating this feature, i.e. obtained by adding another feature to COUNTER and satisfying the interaction requirements. This provides an example of a "good" interaction with COUNTER.

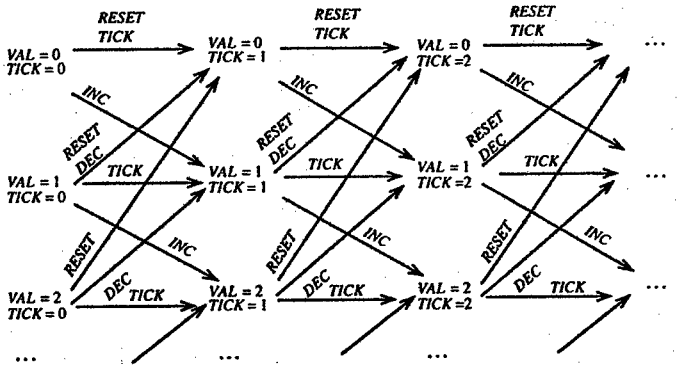
simple system COUNTER_TIME[c] =
data NAT
attrs $VAL, TIME$: *nat*
interface $INC, DEC, RESET, TICK()$
activity $\mathsf{I}: c \xrightarrow{INC} c[VAL+1/VAL][TIME+1/TIME]$

if $VAL > 0$ **then** D: $c \xrightarrow{DEC} c[VAL - 1/VAL][TIME + 1/TIME]$

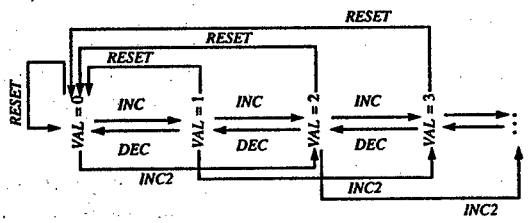
R: $c \xrightarrow{RESET} c[0/VAL][TIME + 1/TIME]$

T: $c \xrightarrow{TICK} c[TIME + 1/TIME]$

Its semantics is essentially the lts graphically depicted below.



As an example of "bad" interaction let us consider the system whose semantics is depicted below. In this case the attribute *VAL* is changed by transitions with a new label (*INC2*).



End example

An interaction requirement for a feature F is a formula of an appropriate logic expressing some property of a generic system obtained by adding other features to F. There are many choices for the logic to use, e.g., first-order, temporal-logic, but also non-standard logics for expressing conditions on the added attributes and label constructors. Here we prefer to allow only interaction requirements having a very precise form, both for methodological (only sensible properties are expressed) and for technical reasons (the formal setting is simple enough but has interesting characteristics). Experiment with more realistic examples will help to find out a good choice.

Here the general form of an interaction requirement is

if $a: s \xrightarrow{l} s'$ **then** *cond*

where a, s, l, s' are variables of the appropriate types, *cond* is a first-order formula on a, s, l and s', built using only the signature of the basic data structures, the operations extracting the value of the attributes from the states and the label constructors.

The following example presents further interaction requirements and shows a case when "action names" differ from labels.

Example 2.3 The possibility of failing as a feature

feature FAIL[c] =
data BOOL
attrs *FAILED*: *bool*
interface *INC*, *DEC*, *RESET*()
activity FI: $c \xrightarrow{INC} c[True/FAILED]$
FD: $c \xrightarrow{DEC} c[True/FAILED]$
if *FAILED* = *False* **then** I: $c \xrightarrow{INC} c$
if *FAILED* = *False* **then** D: $c \xrightarrow{DEC} c$
if *FAILED* = *False* **then** R: $c \xrightarrow{RESET} c$
interaction requirements
if a: $c \xrightarrow{l} c'$ **and** *FAILED* = *True* **then** *FAILED'* = *True*
if a: $c \xrightarrow{l} c'$ **and** (a = I **or** a = D **or** a = R) **then** *FAILED* = *FAILED'*

Notice how the last three rules, which seem of scarce relevance, define the relations between the actions I, D R and the attribute *FAILED*, by imposing that they can be performed only when *FAILED* is false. The interaction requirements mean that once the attribute *FAILED* is true, it cannot become again false and that actions I, D, R do not modify *FAILED*. In this case actions I and FI correspond respectively to the correct and failed execution of the increment command; similarly for D and FD. **End example**

We give below another example of feature which will be used in the following to give examples of interactions.

Example 2.4 A sensor controlling the counter The counter levels over 99 are considered dangerous, and so a warning must be issued in such cases; the attribute *INFORMED* is used to be sure that the warning is issued only once. Transitions with label *NORMAL* put the system back to a normal situation setting the attribute *VAL* to a given value (*NORMAL* is a label constructor parameterized by a natural value).

feature WARNING[c] =
data BOOL, NAT
attrs *VAL*: *nat*
INFORMED: *bool*
interface *WARNING*, *RESET*()
NORMAL(*nat*)
activity if *VAL* $\geq$ 100 **and** *INFORMED* = *False* **then** W: $c \xrightarrow{WARNING} c[True/INFORMED]$
if n < 100 **and** *INFORMED* = *True* **and** *VAL* $\geq$ 100 **then**
BN: $c \xrightarrow{NORMAL(n)} c[False/INFORMED][n/VAL]$
R: $c \xrightarrow{RESET} c[False/INFORMED]$
interaction requirements
"INFORMED is private"
"WARNING is private"
"NORMAL is private"

"*INFORMED* is private" is a shortcut requiring that the attribute *INFORMED* is local and can only be modified as expressed by the rules. The corresponding expanded version is

if $a{:}\,c \xrightarrow{l} c'$ **and** $INFORMED \neq INFORMED'$ **then**
($l = WARNING$ **and** $VAL \geq 100$ **and** $INFORMED = False$ **and** $INFORMED' = True$) **or**
(**exists** n **s.t.** $l = NORMAL(n)$ **and** $n < 100$ **and** $VAL \geq 100$ **and**
$VAL' = n$ **and** $INFORMED = True$ **and** $INFORMED' = False$) **or**
($l = RESET$ **and** $INFORMED' = False$)
Analogously *"WARNING is private"* and *"NORMAL is private"* correspond to
if $a{:}\,c \xrightarrow{WARNING} c'$ **then** $VAL \geq 100$ **and** $INFORMED = False$ **and** $INFORMED' = True$
if $a{:}\,c \xrightarrow{NORMAL(n)} c'$ **then**
$VAL \geq 100$ **and** $n < 100$ **and** $INFORMED = True$ **and** $INFORMED' = False$ **and** $VAL' = n$

The most common interaction requirements have the above forms, just requiring that an action name/an attribute/a label constructor is private. **End example**

2.3 Semantics

Recall that a feature is essentially the description of "parts" of a simple reactive system plus some requirements on other "parts" which can be added to get a complete system. Usually it is designed by first giving the parts and afterwards the requirements. Thus we first provide a semantics, the *basic semantics*, taking into account only the given parts; afterwards, in order to consider the interactions with other features, we provide a second one, the *complete semantics.*

Basic Semantics In our setting, the parts provided by a feature specification result to be a specification of a simple system, and the *basic semantics* of a feature F, denoted by $[\![F]\!]_B$, is just the semantics of the associated simple system specification, $Syst(F)$, determined by the data structures, attributes, label constructors and rules of F.

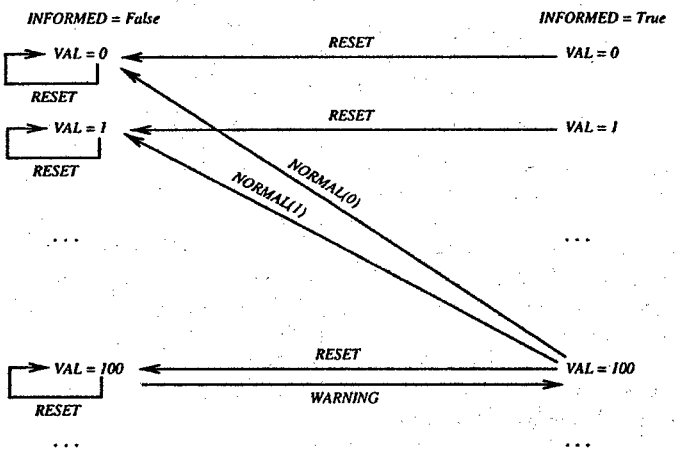
Example 2.5 The basic semantics of COUNTER is just the semantics of the system specification COUNTER_S reported in Ex. 2.1; while the one of FAIL is graphically depicted below.

By looking at the above lts, we can already detect some trouble in FAIL; precisely that the failed system may go on receiving increment and decrement commands. If we intend to give a feature without that behaviour, then we may replace the first two rules with

if $VAL = False$ **then** FI: $c \xrightarrow{INC} c[True/FAILED]$,
if $VAL = False$ **then** FD: $c \xrightarrow{DEC} c[True/FAILED]$;

the new basic semantics is then

The basic operational semantics of WARNING is graphically depicted below.



End example

Semantics The *(complete) semantics* should associate with a feature F a precise (formal) meaning taking into account all its components, thus also the interaction requirements. For us it is the class of all simple systems (lts's modelling them) having at least all parts specified by F and satisfying all its interaction requirements. These systems may have more data, attributes, label constructors and actions than *Syst*(F), and clearly also more transitions (i.e. more activity); technically, their signature is an *extension* of the signature of F.

A most important remark is that we have to allow these systems to *loosely satisfy* the original transition rules of F, in the sense that whenever a rule does not explicitly state that an attribute remain unchanged, then such change may be allowed in the system. All this amounts to adopt a kind of anti-frame assumption. We can provide the following detailed definition.

Def. 2.6 (Complete semantics)

Let F be a feature specification over a signature Σ_F. Then the *complete semantics* of F, denoted by **[F]**, is the class of all M s.t.

- M is an lts over some extension $\Sigma_{F'}$ of Σ_F;
- M satisfies all interaction requirements of F;
- M loosely satisfies all rules of F

  (formally, M loosely satisfies **if** *cond* **then** a: $s \xrightarrow{t} s[x_1/A_1]\ldots[x_k/A_k]$ iff

  for all variable evaluations V,

  if $M, V \models cond$, then

  there exists $\overline{s}$ state of M s.t. $V(a)$: $V(s) \xrightarrow{V(t)} \overline{s}$ and for $j = 1, \ldots, k$ $A_j(\overline{s}) = V(x_j)$.

□

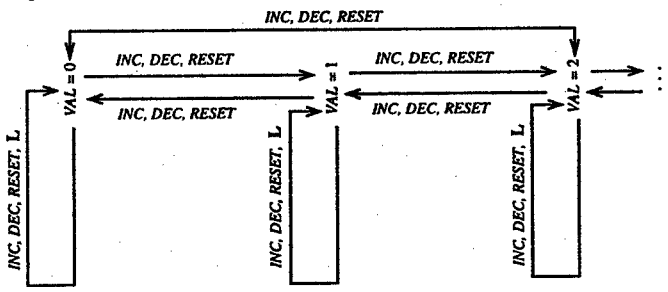
Since the complete semantics usually consists of infinitely many models, in the following section we will illustrate a possible canonical representation (a kind of abstract interpretation) for reasoning about.

2.4 Representing Possible Interactions

Let F be a generic feature specification. We want to analyse F trying to understand its characteristics w.r.t. the interaction with other features; that means to examine $[\![F]\!]$ from a behavioural point of view. We do that by singling out a special system in $[\![F]\!]$, $MAX(F)$, whose behaviour may help to understand the possible interactions; more precisely the one corresponding to the maximum of interactions with other features.

Other features may add to F whatever new attributes and labels and so $MAX(F)$ should have infinitely many attributes and label constructors; but, since we are interested only in the behaviour, the attributes (the actual form of the state) are irrelevant and so $MAX(F)$ has the same attributes of F; while $MAX(F)$ will have just an extra label **L** abstractly representing all labels different from those of F.

Example 2.7 As an example below we present MAX(COUNTER).

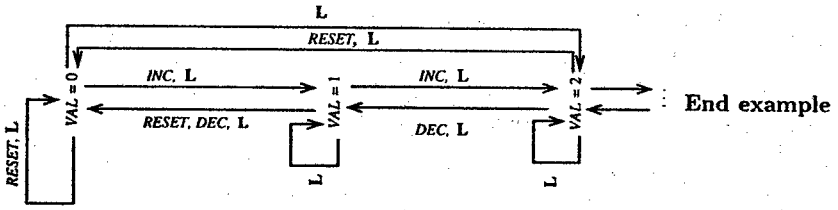


We can see that imposed interaction requirements are weak, since new transitions with label in $\{DEC, INC, RESET\}$ may freely modify *VAL*. Notice how we can immediately see that transitions with new labels cannot modify *VAL*, but recall that does not mean they cannot modify at all the state, in the above representation we are abstracting from possible new attributes.

Consider instead the feature COUNTER′ differing from COUNTER only for the interaction requirement part, which becomes "*INC*, *DEC* and *RESET* are private", corresponding to the axioms

if $a{:}\,c \xrightarrow{RESET} c'$ **then** $VAL' = 0$
if $a{:}\,c \xrightarrow{INC} c'$ **then** $VAL' = VAL + 1$
if $a{:}\,c \xrightarrow{DEC} c'$ **then** $VAL > 0$ **and** $VAL' = VAL - 1$.

COUNTER′ seems more sensible as it is shown by the behaviour of MAX(COUNTER′).



End example

We can now propose the following definition of $MAX(\mathrm{F})$.

Def. 2.8 Let F be a feature specification.

- $Sig^{+}(\mathrm{F})$ is the feature signature defined by adding to $Sig(\mathrm{F})$ a constant label constructor **L**.
- $\mathcal{X}$ is the set of all $Sig^{+}(\mathrm{F})$-lts's M s.t.
  - the data structure part of M coincides with that of F;
  - the states of M are those of $[\![\mathrm{F}]\!]_{\mathrm{B}}$;
  - the labels of M are those of $[\![\mathrm{F}]\!]_{\mathrm{B}}$ plus a special one **L**;
  - the action names of M are those of $[\![\mathrm{F}]\!]_{\mathrm{B}}$ plus a special one **A**;
  - the transition relation of M (denoted by $\rightarrow^{M}$) is s.t. $M \models \phi$ for all interaction requirements ϕ of F.
- $MAX(\mathrm{F})$ is the element of $\mathcal{X}$ s.t. $\rightarrow^{MAX(\mathrm{F})} = \cup_{M \in \mathcal{X}} \rightarrow^{M}$. □

Notice that the existence of $MAX(\mathrm{F})$ is permitted by the particular form of the interaction requirements (see Sect. 2.2), which express in some sense only safety properties on single transitions. For example an interaction requirement like

$a{:}\, s \xrightarrow{L1} s'$ **iff not** $a{:}\, s \xrightarrow{L2} s'$

would prevent the existence of the maximum element intended as above.

Also notice that, if F is consistent, then $MAX(\mathrm{F}) \in [\![\mathrm{F}]\!]$, since any rule ρ compatible with the interaction requirements is loosely satisfied by $MAX(\mathrm{F})$.

The methodological role of $MAX(\mathrm{F})$ for reasoning about interaction is clarified by an appropriate notion of behavioural simulation: $MAX(\mathrm{F})$ *abstractly simulates* all possible behaviours of the systems in the complete semantics of F. In a sense this explain why $MAX(\mathrm{F})$ can be seen as an abstract interpretation of the specification of F.

The intuitive idea of abstract simulation is the following.

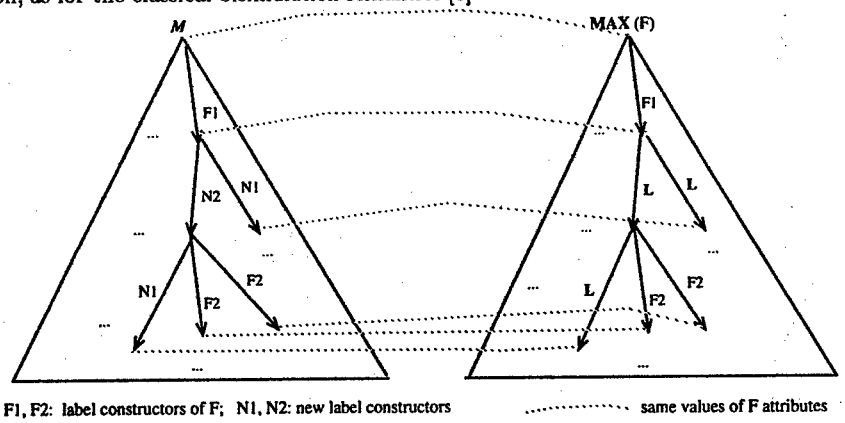
$M \in [\![\mathrm{F}]\!]$ is *abstractly simulated* by $MAX(\mathrm{F})$ iff for any state s of M there exists a state ms of $MAX(\mathrm{F})$ s.t. s is abstractly simulated by ms; where

s is *abstractly simulated* by ms iff

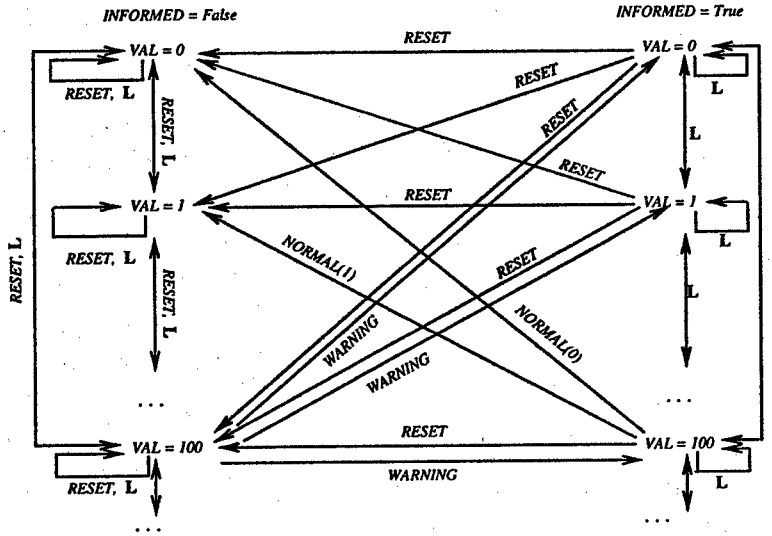
- the value of the attributes of F are the same in s and ms;
- if $s \xrightarrow{l} s'$ with l built by a label constructor of F, then there exists ms' s.t. $ms \xrightarrow{l} ms'$ and s' is abstractly simulated by ms';

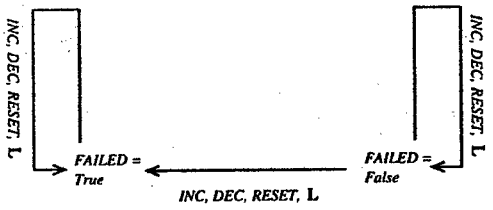
- if $s \xrightarrow{l} s'$ with l built by a label constructor not of F, then there exists ms' s.t. $ms \xrightarrow{\mathrm{L}} ms'$ and s' is abstractly simulated by ms'.

The intuition, graphically presented below, can be refined to a completely rigorous definition, as for the classical bisimulation semantics [6].



Example 2.9 We present below in a graphic way MAX(WARNING) and MAX(FAIL).





End example

3 Features and Interactions

3.1 Compatibility and Composition

A careful scrutiny of concrete examples shows that feature composition is not at all an absolute notion, in the sense that it depends very much on the application and on the way features are represented. Thus our approach is essentially methodological, though instantiated on our particular feature description formalism.

We factorize the problem in two ways: first we support a notion of compatibility, preliminary to the definition of feature composition; second we deal with compatibility and composition componentwise w.r.t. the structure of the feature specifications. The method we advocate is to build feature specifications, perhaps by means of some forth and back process, in a way that composition be possible and achieve the intended goals (requirements). In the following section we will examine from a behavioural viewpoint the composition of features, in order to introduce another criterion for reasoning about feature composition and checking whether it corresponds to the intended goals.

Let us assume to have the feature specifications F_1 and F_2, with components $DATA_1$, $ATTRS_1$, $LABS_1$, $RULE_1$, IR_1 and $DATA_2$, $ATTRS_2$, $LABS_2$, $RULE_2$, IR_2 respectively.

We will use $\oplus$ to denote the composition operation for features and also the those for the subcomponents.

Data The data parts are compatible iff $DATA_1$ and $DATA_2$ have disjoint signatures or share the common part, a sub data-structure.

Attributes The attribute components are compatible iff all attributes with the same name in both features have the same result type.

$ATTRS_1 \oplus ATTRS_2 = ATTRS_1 \cup ATTRS_2$

Label constructors The interface components are compatible iff all label constructors with the same name in both features have the same number of arguments, of the same type and in the same order.

$LABS_1 \oplus LABS_2 = LABS_1 \cup LABS_2$

Rules Since rules correspond to partial descriptions of action capabilities we need in the feature specifications a mechanism for deciding which rules $r_1 \in RULE_1$ and $r_2 \in RULE_2$

describe parts of the same action capability, and thus have to be composed. Several criteria are possible, as:

1. no pair (i.e., rules correspond to complete action capability description);
2. those having the same label constructor;
3. those belonging to the same action.

To compose rules with the same action name is the most general case; indeed the other cases may be recovered by appropriately choosing the action names (case 1) corresponds to have all action names of F_1 different from those of F_2, and case 2) to use the label constructors as action names).

Moreover, we need to define when two rules (partial action descriptions) are compatible. Then the rule parts are compatible iff each pair of rules to be composed is compatible.

In the following we assume that the rules of the two features:

- use only variables to denote the new values of updated attributes (in the update parts) and the arguments of the label constructors, e.g., $a\colon s \xrightarrow{L(3+x,0)} s[4*B/E]$ could be equivalently written

  if $y_1 = 3+x$ **and** $y_2 = 0$ **and** $e = 4*B$ **then** $a\colon s \xrightarrow{L(y_1,y_2)} s[e/E]$.
- use both the same variables to denote the new values of shared attributes and the arguments of the shared label constructors;
- do not share free variables except those denoting the new values of shared attributes and the arguments of the shared label constructors.[1]

Let

$r_1 =$ **if** $cond_1$ **then** $\mathsf{ACT}_1\colon s \xrightarrow{L_1(y_1^1,\ldots,y_{m_1}^1)} s[x_1^1/A_1^1]\ldots[x_{k_1}^1/A_{k_1}^1]$

and

$r_2 =$ **if** $cond_2$ **then** $\mathsf{ACT}_2\colon s \xrightarrow{L_2(y_1^2,\ldots,y_{m_2}^2)} s[x_1^2/A_1^2]\ldots[x_1^2/A_{k_2}^2]$

be two rules;

Single rule compatibility r_1 and r_2 are *compatible* iff $\mathsf{ACT}_1 = \mathsf{ACT}_2$ and $L_1 = L_2$ (thus also $y_1^1,\ldots,y_{m_1}^1 = y_1^2,\ldots,y_{m_2}^2$);

Single rule composition If r_1 and r_2 are compatible, then $r_1 \oplus r_2 =$

if $cond_1$ **and** $cond_2$ **then**

$\mathsf{ACT}_1\colon s \xrightarrow{L_1(y_1^1,\ldots,y_{m_1}^1)} s[x_1^1/A_1^1]\ldots[x_{k_1}^1/A_{k_1}^1][x_1^2/A_1^2]\ldots[x_1^2/A_{k_2}^2]$.

Note that $r_1 \oplus r_2$ may be a null rule (i.e. a rule which does not generate any transition, since its premises cannot be satisfied). For example the composition of two rules updating both the same attribute with different values is a null rule, since $cond_1$ **and** $cond_2$ contains $x = t_1$ **and** $x = t_2$ with t_1 and t_2 ground terms representing different elements of the data structure.

[1] This restriction may be overcome by using action names with parameters (and so a mechanism for deciding whether two such variables refer to the same quantity)

Rules compatibility $RULE_1$ and $RULE_2$ are *compatible* iff for all $r_1 \in RULE_1$ and $r_2 \in RULE_2$ about the same action, r_1 and r_2 are compatible.

Rules composition If $RULE_1$ and $RULE_2$ are *compatible* then we define

$RULE_1 \oplus RULE_2 =$

$\{r_1 \in RULE_1 \mid \nexists\, r_2 \in RULE_2$ about the same action $\} \cup$

$\{r_2 \in RULE_2 \mid \nexists\, r_1 \in RULE_1$ about the same action $\} \cup$

$\{r_1 \oplus r_2 \mid r_1 \in RULE_1, r_2 \in RULE_2$ about the same action $\}$.

Interaction requirements If the components of F_1 and of F_2 related to the system specification are compatible as defined above, then we compose them just by composing data, attributes, labels and rules, getting a simple system specification denoted by $Syst(F_1) \oplus Syst(F_2)$.

IR_1 and IR_2 are *compatible* iff the composition of the system specification parts, $Syst(F_1) \oplus Syst(F_2)$,
satisfies IR_1 and IR_2, i.e.:

$Syst(F_1) \oplus Syst(F_2) \models IR_1 \cup IR_2$

$IR_1 \oplus IR_2 = IR_1 \cup IR_2$.

Clearly if $IR_1 \cup IR_2$ is inconsistent, then the two features are not compatible.

Feature compatibility F_1 and F_2 are compatible iff all their components are so.

Feature composition If F_1 and F_2 are compatible, then $F_1 \oplus F_2$ is the feature specification whose components are the compositions of the corresponding ones of F_1 and of F_2.

Prop. 3.1 Properties of $\oplus$

Given two compatible feature specifications for simple systems F_1 and F_2; we have that

1. $[\![F_1 \oplus F_2]\!] \subseteq [\![F_1]\!] \cap [\![F_2]\!]$;
2. $\oplus$ is commutative and associative. □

Instead, $\oplus$ is not idempotent.

Let us illustrate the concepts introduced so far by means of two examples; the second also illustrate the kind of backward adjustment we have in mind.

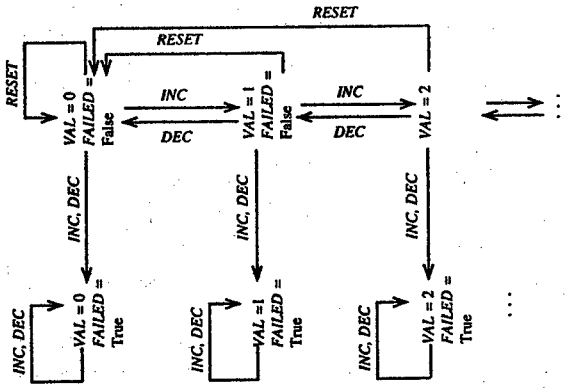
Example 3.2 Composing COUNTER **and** FAIL (defined in Ex. 2.2 and 2.3) The data, the attributes and the interfaces of the two features are trivially compatible; for the actions R, I, D there are rules in both features, but they are compatible, and so we can compose the two features at the system level.

simple system $Syst(\text{COUNTER}) \oplus Syst(\text{FAIL})[c] =$
data BOOL, NAT
attrs *VAL: nat*
FAILED: bool

interface $INC, DEC, RESET()$
activity **if** $FAILED = False$ **then** I: $c \xrightarrow{INC} c[VAL+1/VAL]$
if $VAL > 0$ **and** $FAILED = False$ **then** D: $c \xrightarrow{DEC} c[VAL-1/VAL]$
if $FAILED = False$ **then** R: $c \xrightarrow{RESET} c[0/VAL]$
FI: $c \xrightarrow{INC} c[True/FAILED]$
FD: $c \xrightarrow{DEC} c[True/FAILED]$

Such system, whose behaviour is graphically presented below, verifies the interaction requirements of both F_1 and F_2, and so the two features may be composed.

Notice the role played by the action names in this composition, the failed executions of the increment/decrement commands do not change the attribute *VAL*.

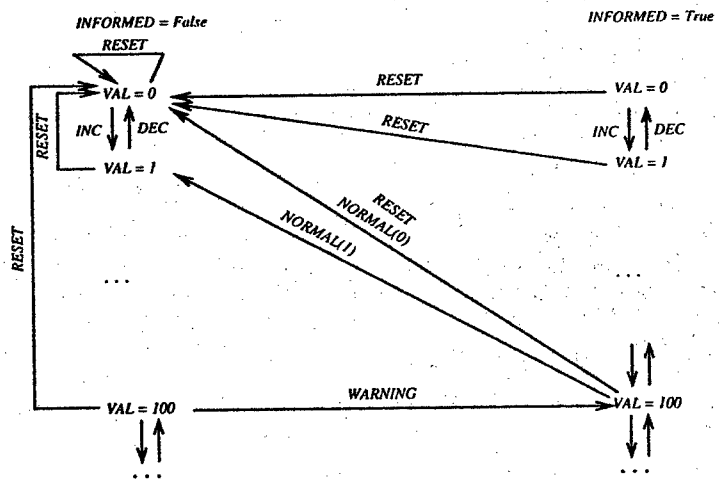


End example

Example 3.3 Composing COUNTER and WARNING (defined in Ex. 2.2 and 2.4) They are compatible at system level, and the composition of their system parts is

simple system *Syst*(COUNTER) ⊕ *Syst*(WARNING)[*c*] =
data BOOL, NAT
attrs *VAL*: *nat*
INFORMED: *bool*
interface $INC, DEC, RESET, WARNING()$
$NORMAL(nat)$
activity I: $c \xrightarrow{INC} c[VAL+1/VAL]$
if $VAL > 0$ **then** D: $c \xrightarrow{DEC} c[VAL-1/VAL]$
R: $c \xrightarrow{RESET} c[0/VAL][False/INFORMED]$
if $VAL \geq 100$ **and** $INFORMED = False$ **then** W: $c \xrightarrow{WARNING} c[True/INFORMED]$
if $n < 100$ **and** $INFORMED = True$ **and** $VAL \geq 100$ **then**
BN: $c \xrightarrow{NORMAL(n)} c[False/INFORMED][n/VAL]$

whose behaviour is graphically represented by

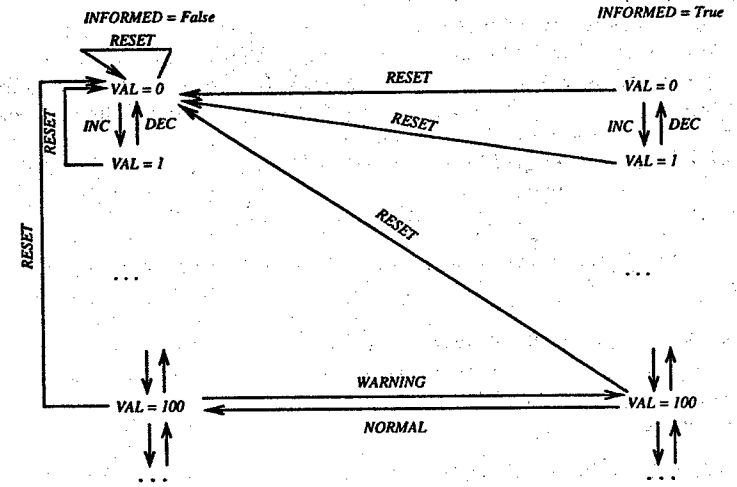


However this system does not satisfy the interaction requirements of COUNTER; indeed transitions with labels $NORMAL(n)$ may change the attribute VAL; instead the interaction requirements of WARNING are satisfied.

Consider now another feature WARNING′ obtained by WARNING by making $NORMAL$ a label without parameters and changing the corresponding rule into

if $INFORMED = True$ **and** $VAL \geq 100$ **then** BN: $c \xrightarrow{NORMAL} c[False/INFORMED]$;

WARNING′ and COUNTER are compatible, and the operational semantics of their composition is



End example

Example 3.4 Composing FAIL and WARNING′ The two features are compatible at the system level, resulting in

simple system $Syst(\text{FAIL}) \oplus Syst(\text{WARNING}')[c] =$
data BOOL, NAT
attrs *FAILED*, *INFORMED*: *bool*
VAL: *nat*
interface *INC*, *DEC*, *RESET*, *WARNING*, *NORMAL*()
activity FI: $c \xrightarrow{INC} c[True/FAILED]$
FD: $c \xrightarrow{DEC} c[True/FAILED]$
if *FAILED* = *False* **then** I: $c \xrightarrow{INC} c$
if *FAILED* = *False* **then** D: $c \xrightarrow{DEC} c$
if *FAILED* = *False* **then** R: $c \xrightarrow{RESET} c[False/INFORMED]$
if $VAL \geq 100$ **and** *INFORMED* = *False* **and** $VAL \geq 100$ **then**
W: $c \xrightarrow{WARNING} c[True/INFORMED]$
if *INFORMED* = *True* **and** $VAL \geq 100$ **then** BN: $c \xrightarrow{NORMAL} c[False/INFORMED]$

and this system satisfies the interaction requirement of both, and so they are compatible.
End example

3.2 Feature Interaction

As already suggested, when organizing a system by features, it is of paramount importance to have a clear picture of the variations which may occur when adding features. Here we try to single out some basic criteria for reasoning about feature interactions.

Together with many others (but not all) by "interaction of feature F_2 on F_1" (or "F_2 interacts with F_1") we mean that the "part of $F_1 \oplus F_2$ due to F_1 is not as specified by F_1". Here we try to provide a rigorous background for this meaning of interaction. In the following we try to propose the main concepts, illustrated by examples; but we do not pretend to propose a theory, for the moment.

First of all we need to define what is the part due to F_1 in $[\![F_1 \oplus F_2]\!]_B$. We assume that it is the "projection of $[\![F_1 \oplus F_2]\!]_B$ on the signature $Sig(F_1)$", i.e. the lts having only the actions, attributes and labels in $Sig(F_1)$ and whose transitions are obtained by those of $[\![F_1 \oplus F_2]\!]_B$ dropping those whose labels or actions are not in $Sig(F_1)$ and projecting the states on the $Sig(F_1)$ attributes of the others. We may have different concepts of interaction between features depending on how we compare the activity of $[\![F_1]\!]_B$ w.r.t. that of $[\![F_1 \oplus F_2]\!]_B$ projected on F_1; here we consider:

a) atomic(-level) interaction, if we look at the single transitions (corresponding to atomic activities);

b) behaviour(-level) interaction, if we consider whether $[\![F_1]\!]_B$ abstractly simulates all behaviours of $[\![F_1 \oplus F_2]\!]_B$ projected on F_1 and vice versa, where "abstract simulation" has been introduced in Sect. 2.4, and here we abstract w.r.t. the attributes and labels not of F_1.

If there are no interactions following **b)**, then there are no interactions following **a)** too.

Atomic Interaction A feature F_2 atomically interacts with F_1 whenever the transitions of $[\![F_1]\!]_B$ are different from those of $[\![F_1 \oplus F_2]\!]_B$ projected onto the attributes and the labels of F_1. There are various reasons for such transitions to be different, as:

- the premises of a rule of F_1 become more restrictive when it is composed with a rule of F_2;
- a rule of F_1 when it is composed with one of F_2 may update attributes of F_1 previously not modified;
- new rules about new action names but with F_1 labels and modifying the F_1 attributes are added by F_2.

Let F_1 and F_2 be two compatible feature specifications. We say that F_2 *atomically-interacts with* F_1 iff $\rightarrow^{[F_1]_B}$ is different from the set of transitions $\overline{s} \xrightarrow{l} \overline{s'}$ s.t.

- $s \xrightarrow{l} s' \in \rightarrow^{[F_1 \oplus F_2]_B}$,
- l is built by a label constructor of F_1 and
- $\overline{s}, \overline{s'}$ are the projections of s, s' onto the attributes of F_1.

F_1 and F_2 *are atomically-independent* iff F_1 does not atomically-interacts with F_2 and F_2 does not atomically-interact with F_1.

It is not true that any case of interaction intended in this way is negative and must be prohibited; for example, think of a feature for a telephone system which offers a discount whenever some particular green numbers are called (transitions which do not change the debt of a user, now decrease it).

Example 3.5 COUNTER and FAIL are not atomically-independent; indeed by looking at Ex. 2.1 and Ex. 3.2 we can see that FAIL has added transitions with labels *INC* and *DEC* which do not modify the attribute *VAL*.

Instead by looking also at Ex. 3.3 we can see that COUNTER and WARNING' are atomically-independent. **End example**

This view of interaction has very nice properties.

Indeed, it is possible to give extremely general and almost syntactic sufficient condition for ensuring that two features are atomically independent (where "extremely general" means that if F_1 and F_2 are atomically independent, then we can modify F_1 and F_2 to get two new feature specifications F_1' and F_2' so that their composition is the same of F_1 and of F_2 and F_1' and F_2' satisfy the sufficient conditions; "almost syntactic" means that the conditions are syntactic, except for what concerns the data elements). Thus it should be possible to develop tools for automatically checking atomic independence.

Furthermore atomic interaction may be disciplined by interaction requirements made by the subset of the safety formulae as presented in Sect. 2.

if $a: s \xrightarrow{l} s'$ **then** *cond*

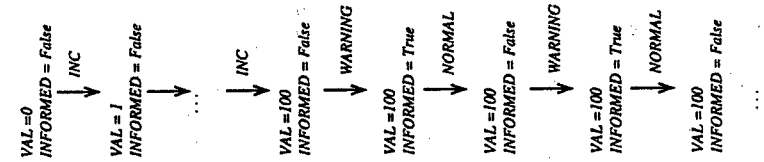
where a does not occur in *cond*. Indeed, for each pair of compatible features F_1 and F_2 s.t. F_2 atomically interacts with F_1, there exists a formula ϕ of the above type s.t. added to the interaction requirements of F_1 makes F_1 and F_2 not compatible.

Behavioural Interaction A feature F_2 behaviourally interacts with F_1 whenever either $[\![F_1]\!]_B$ is not abstractly simulated by $[\![F_1 \oplus F_2]\!]_B$ or vice versa, where in this case "abstractly" means to forget the attributes and the label constructors not of F_1.

F_1 and F_2 *are behaviourally-independent* iff F_1 does not behaviour-interact with F_2 and F_2 does not behaviour-interact with F_1.

It is not true that any case of interactions intended in this way are negative and must be prohibited; for example a feature stating that all calls made on Sunday are free results in a case of behavioural interaction; indeed the handling of the free Sunday call is an added piece of behaviour, while handling of the paid Sunday calls has been removed.

Example 3.6 The feature WARNING' behaviourally interacts with COUNTER; indeed by looking at Ex. 3.3 we see in $[\![\text{COUNTER} \oplus \text{WARNING'}]\!]_B$ the behaviour



which cannot be abstractly simulated by any behaviour of $[\![\text{COUNTER}]\!]_B$ (see Ex. 2.1), since when *VAL* is equal to 100 any transition of $[\![\text{COUNTER}]\!]_B$ modifies its content. **End example**

Unfortunately it is not so easy to find sufficient condition for two features being behaviourally independent general enough, and we have to find which are the formulae to be used as interaction requirements to be able to discipline this kind of interaction, i.e. as for atomic interaction, which are the formulae that added to the interaction requirements part may make two compatible behaviourally interacting features incompatible.

4 Conclusion and Related Work

We have illustrated a rather general framework for feature-oriented development. Somewhat differently than in other approaches, our aim is to provide a flexible discipline for handling features, more than just checking the interactions. The flexibility is provided by factorizing the specification development: first, by means of the representation of the complete semantics we can check wether our feature specification which includes interaction requirements, corresponds to our intuitive understanding; second, when adding some new feature we may adjust its specification, by checking its capabilities and reasoning about the resulting composition.

In our opinion the general framework should be adapted to the particular domain specific application, as it is supported by the work of P. Zave on telephone systems [11].

It is worthwhile mentioning that the useful graphical representations are really possible not only for the toy examples considered in the paper; indeed there is a way of presenting graphically design specifications for reactive and concurrent sytems (see [8]), which is adjustable to the case of features, as we plan to do in some further work.

As for the automatic generation of the graphical representations, it does not sem out of the state-of-the-art, though not yet explored by us.

Together with improving the graphical representation, our ongoing work aims at dealing with features for structured systems, i.e. systems made of subsystems; in other words we want to have at hand both component and feature modularity.

Recently some papers trying to study features and their interactions on a formal ground have started to appear, but none of them presents something of similar to our "interaction requirements". Among them we recall [2], presenting feature specifications based on logical formulae conceptually similar to our rules, but their idea of "feature composition" and of "interaction" is really less flexible than our (e.g., using our terminology transitions are composed only when have the same label, and interaction is just atomic interaction). In e.g., [3] Bredereke, trying to give a formal view of features and interactions, considers the importance of the behavioural aspects. Also [4] presents a formal based treatment of features for telecommunication systems, but at a more concrete level (i.e. more oriented towards the implementation) and so it is difficult it to fully relate to our work.

Prehofer considers both methodological aspects as in [7] and formal aspects, as in [5], where he presents an approach based on transition systems (diagrams); but differently from our work, for him to add a feature to a system means to refine graphically a part of the diagram specifying it. It is interesting to note that our framework may offer a formal counterpart to part of his work in [7] including the "lifters", i.e. feature modifiers for helping to resolve feature interactions.

References

[1] E. Astesiano and G. Reggio. Labelled Transition Logic: An Outline. Technical Report DISI–TR–96–20, DISI – Università di Genova, Italy, 1996.

[2] J. Blom, R. Bol, and L. Kempe. Automatic Detection of Feature Interactions in Temporal Logic. Technical Report 95/61, Department of Computer Systems, Uppsala University, 1995.

[3] J. Bredereke. Formal Criteria for Feature Interactions in Telecommunications Systems. In J. Norgaard and V. B. Iversen, editors, *Intelligent Networks and New Technologies.* Chapman & Hall, 1996.

[4] M. Faci and L. Logrippo. Specifying Features and Analyzing their Interactions in a LOTOS Environment. In L.G. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems (Proc. of the 2nd International Workshop on Feature Interactions in Telecommunications Systems, Amsterdam)*, pages 136–151. IOS Press, 1994.

[5] C. Klein, C. Prehofer, and B. Rumpe. Feature Specification and Refinement with State Transition Diagrams. In P. Dini, editor, *Fourth IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems.* IOS-Press, 1997.

[6] R. Milner. *Communication and Concurrency.* Prentice Hall, London, 1989.

[7] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of ECOOP'97*, number 1241 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1997.

[8] G. Reggio and M. Larosa. A Graphic Notation for Formal Specifications of Dynamic Systems. In J. Fitzgerald and C.B. Jones, editors, *Proc. FME 97 - Industrial Applications and Strengthened Foundations of Formal Methods*, number 1313 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1997.

[9] M. Wirsing. Algebraic Specifications. In J. van Leeuwen, editor, *Handbook of Theoret. Comput. Sci.*, volume B, pages 675–788. Elsevier, 1990.

[10] P. Zave. Feature interactions and formal specifications in telecommunications. *Computer*, 26(8):20–29, 1993.

[11] P. Zave. Calls considered harmful and other observations: A tutorial onfftelephony. In T. Margaria, editor, *Second International Workshop onffAdvanced Intelligent Networks '97*, 1997.

Merging Changes to Software Specifications*

V. Berzins
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Abstract

We propose a model of software changes and a method for combining changes to a software specification. This work was motivated by the desire to provide automated decision support for the evolution of software prototypes. We define a behavioral refinement ordering on software specifications and indicate how this structure can be used to automatically combine several changes to a specification. A set of examples illustrates the ideas.

1 Introduction

Changing software without damaging it is difficult and accounts for the bulk of software-related costs. This issue is particularly prominent in the context of software prototyping, where requirements, specifications, and designs are undergoing radical and repeated change, under constraints of low cost and rapid response. In this context teams often explore changes to different aspects of a system concurrently, and may develop prototypes of several competing formulations simultaneously, to obtain user guidance about the benefits and drawbacks of different alternatives. When the preferred alternatives are clear, we must consistently combine the changes to the system specification corresponding to the preferred alternative for each aspect of the system that has been explored.

This paper presents a formal model and a method for addressing this problem in the context of black-box specifications for systems. We address specifications expressed in logic, using a notation for system specification that has been designed to support development of large and complex systems [5]. We explore the problem in the context of prototyping because it is a promising way to address the main source of system faults, namely requirements errors [23]. Evolutionary prototyping provides an efficient approach to formulating accurate software requirements [20].

The focus of the current work is the evolution of proposed specifications and prototype designs. Much of the previous work on changes to software has focused on meaning-preserving transformations [2, 15, 17, 27]. However, it has been recognized that in realistic

*This research was supported in part by the National Science Foundation under grant number CCR-9058453, by the Army Research Office under grant number 30989-MA, and by the Army AI Center under grant number 6GNPG00072.

contexts, many changes do *not* preserve the observable behavior of the system [28]. Most of the work on the area of meaning-changing transformations has been concerned with classifying the types of semantic modifications that are used in practice [13, 12, 16]. We investigate the relationships between different versions of the specifications and propose an abstract model of the design history to provide a more formal model for understanding the details of this subject.

Modeling the design history can enhance the prototyping process by capturing the conceptual dependencies in a design. A properly structured derivation of a specification can highlight the structure of the design decisions leading to the proposed system, which can be used to record and guide systematic exploration of the design space. Such a representation is necessary if we are to develop software tools for managing this process and extracting useful information from the design history. These tools should help coordinate the efforts of analysts and designers faced with a changing set of requirements, to avoid repeated effort and inconsistent parallel refinements, and to aid the designers in combining design choices from different branches of a parallel exploration of the design space.

In larger prototyping efforts, several explorations of the requirements that are focused on distinct aspects of the system may proceed in parallel. In such cases, the lessons learned from different branches of the effort must be combined and integrated. This is a specification-level instance of the software change-merging problem [8]. Solutions to this problem can also be used to propagate improvements to all affected versions.

The rest of the paper is organized as follows. Section 2 defines a model of software changes and a behavioral refinement ordering for software specifications. Section 3 discusses change merging for specifications and indicates how merged versions can be constructed. Section 4 presents some examples. Section 5 contains conclusions.

2 Software Changes

To formalize changes to black-box descriptions of systems, we must consider what are the externally observable attributes of a system and how the attributes of different versions are related.

2.1 Attributes of System Behavior

We characterize changes to a system specification in terms of three orthogonal attributes of a system: its vocabulary, its behavior, and its granularity [22]. These concepts are reviewed below.

- The *vocabulary* of a system is the set of all external stimuli recognized by the system.
- The *granularity* of a system is the set of all internal stimuli recognized by the system.
- The *behavior* of a system is the set of all possible traces for the system relative to a given vocabulary and granularity.

Each of these three attributes is a set, and is subject to an ordering induced by the subset relation. The resulting partially ordered set becomes a Boolean algebra under the

set union, set intersection, and set complement operations. As explained in Section 3, this structure can support a formal model of software change merging.

If we restrict primitive changes to be monotonic and to affect just one of the three attributes listed above, we get the classification of primitive changes shown in Figure 1, which is repeated from [22].

The symbol A_S represents the attribute A of the original system S, and $A_{S'}$ represents the attribute A of the modified system S'.

	Effect of Change	
Attribute A	$A_S \subset A_{S'}$	$A_S \supset A_{S'}$
Vocabulary	extending	contracting
Granularity	refining	abstracting
Behavior	relaxing	constraining

Figure 1: Types of Changes

A decomposition of the chronological evolution history into primitive substeps conforming to these restrictions enables the rearrangement of a sequential derivation containing meaning-modifying changes into a tree-like rooted directed acyclic graph whose paths consist solely of meaning-preserving changes that add information via compatible vocabulary extensions, granularity refinements, or behavior constraints [11]. We propose this mechanism as a concrete means to document software as if it had been developed using a rational process [24], and conjecture that such structures will be useful for choosing demonstration scenarios, guiding requirements reviews, and summarizing past history for analysts formulating the next version.

A conceptual derivation history is a simplified version of the chronological history of an evolving system that includes only the decisions that were not undone in later steps. We model conceptual derivation histories as graphs whose nodes represent versions and whose arcs represent monotonic changes that add new capabilities or constraints. An idealized prototype evolution process should steadily strengthen the requirements in this sense, until they become acceptable to the users. In practice the path is often less direct. However, a reconstructed direct path in which each step strictly strengthens the requirements should provide a useful summary of the relevant parts of the evolution of the requirements. An example can be found in [11].

2.2 The Behavioral Refinement Ordering

Change merging depends on an ordering with a specialized algebraic structure, usually either a Boolean or Brouwerian algebra [8]. We propose a behavioral refinement ordering $\sqsubseteq$ on software specifications, defined as follows:

$$\begin{aligned} p \sqsubseteq q \;\Leftrightarrow\; & \mathit{vocabulary}(p) \subseteq \mathit{vocabulary}(q)\,\& \\ & \mathit{granularity}(p) \subseteq \mathit{granularity}(q)\,\& \\ & \mathit{behavior}(p) \supseteq \mathit{projection}(\mathit{behavior}(q), \mathit{vocabulary}(p) \cup \mathit{granularity}(p)) \end{aligned}$$

The vocabulary, granularity, and behavior of a specification are defined in Section 2. The projection is needed to ensure that we are comparing just the corresponding parts of

the two behaviors; it removes all events in traces of q that are outside the vocabulary and granularity of p. The ordering $p \sqsubseteq q$ means that q satisfies the specification of p. From the point of view of a user q is just as good as p, and it may be strictly better if it provides some services that p does not. Enhancements can occur if q responds to additional external stimuli, its behavior is specified at a more detailed level of abstraction, or its behavior is subject to stricter constraints.

We would like to separate the effects of changes to orthogonal attributes of the system as much as possible, so that these independent changes can be automatically re-combined in different combinations. The problem of automatically combining different versions of programs has been formally studied in several different contexts [9, 10, 7, 8, 6, 25, 3], and has been informally discussed in terms of the development of requirements in [14], where the independence of elaborations was assessed manually. However, the problem has not yet been solved completely, particularly for requirements.

We make a step towards automating the detection of independent elaborations by proposing a formal model for refinement structures. There is potential for parallel elaboration whenever the refinement ordering can be decomposed in a cross product structure, because different components of the cross product can be refined independently. For example, this is often the case for changes to different messages in a system. Interactions between messages can occur via invariants associated with state models of machines or instance models of types.

Previous methods for software change merging have assumed that the vocabulary is fixed and common to all versions to be merged. The model proposed here is a possible basis for extending some previous work on merging [7, 3] to cases where the vocabulary changes. Such an extension adopts an open and extensible view of the vocabulary: the behavior of a system whose vocabulary does not contain a given stimulus is considered equivalent to the behavior of a modified system that extends its vocabulary with the extra stimulus and leaves its response to that stimulus undefined and unconstrained. This is appropriate for requirements exploration and prototyping, although it is not consistent with the closed-world view typically adopted in software products, where requests outside the vocabulary are expected to produce error messages and have no other effect. Section 3 sketches some of the main ideas for this extension.

3 Combining Changes

The Boolean algebra structure of the vocabulary, granularity, and behavior of a specification identified in Section 2 implies that the usual formulation of the change merging operation can be applied in the context of changes to software specifications. If A, B, and C are specifications, the result of combining the change from B to A with the change from B to C is denoted by $A[B]C$, which is defined as follows.

$$A[B]C = (A - B) \sqcup (A \sqcap C) \sqcup (C - B)$$

Here $\sqcup$ denotes the least upper bound and $\sqcap$ denotes the greatest lower bound with respect to the ordering defined in Section 2.2. The difference is defined by

$$A - B = A \sqcap \overline{B}$$

where the bar denotes the complement operation. This operation is well defined for any Boolean algebra; in the special case of sets ordered by $\subseteq$, it is the set difference operation.

The interpretations of the above Boolean operations for different aspects of software specifications are summarized in Figure 2. Since it is common to represent sets of behaviors by logical assertions representing postconditions, we include the postcondition representation as well.

Aspect	Operation			
	$X \sqsubseteq Y$	$X \sqcup Y$	$X \sqcap Y$	$X - Y$
Vocabulary	$X \subseteq Y$	$X \cup Y$	$X \cap Y$	$X - Y$
Granularity	$X \subseteq Y$	$X \cup Y$	$X \cap Y$	$X - Y$
Behavior	$X \supseteq Y$	$X \cap Y$	$X \cup Y$	$\overline{Y - X}$
Postcondition	$X \Leftarrow Y$	$X \wedge Y$	$X \vee Y$	$X \vee \neg Y$

Figure 2: Concrete Interpretations of Abstract Operations

The set inclusions in the definition of the specification refinement ordering (see section 2.2) go in the opposite direction for the system behavior than for the vocabulary and the granularity. This is reflected in the interpretations of the Boolean operations for those aspects. Since the specification refinement ordering is derived from the orderings of the three different aspects according to the usual ordering construction for a cross product domain, all of the operations extend componentwise. This implies that we can compute change merges for the three aspects independently, according to the interpretations summarized in Figure 2.

4 Examples of Combining Changes to Specifications

Some examples illustrate the effects of the definitions presented in the previous section. Suppose we represent vocabularies as sets of messages. Then the combination of the change that removes the message m_2 from the starting vocabulary $\{m_1, m_2\}$ and the changes that adds m_3 to the same starting vocabulary is calculated as follows:

$$\begin{aligned} &\{m_1\}[\{m_1, m_2\}]\{m_1, m_2, m_3\} \\ &= (\{m_1\} - \{m_1, m_2\}) \cup (\{m_1\} \cap \{m_1, m_2, m_3\}) \cup (\{m_1, m_2, m_3\} - \{m_1, m_2\}) \\ &= \{m_1, m_3\} \end{aligned}$$

The corresponding calculations on postconditions representing behaviors may be bit less intuitive. If P, Q, and R are assertions representing postconditions, we can apply the general definition and simplify to give the following rule:

$$P[Q]R \;=\; (P \vee \neg Q) \wedge (P \vee R) \wedge (R \vee \neg Q) \;=\; (P \vee R) \wedge (Q \Rightarrow P) \wedge (Q \Rightarrow R)$$

We illustrate the consequences of this rule for some common change patterns. Suppose that a, b, and c are three assertions representing postconditions in the specification of the behavior of a system in response to a given stimulus.

The combination of two different constraining changes to a behavior includes both constraints:

$$(a \wedge b)[b](b \wedge c) = (a \wedge b \wedge c)$$

The first changes adds the constraint a to the postcondition b of the base version and the second change adds a different constraint c. The original constraint and both of the new ones are present in the combination.

The combination of two relaxing changes loosens both of the affected constraints:

$$a[a \wedge b]b = a \vee b$$

Note that the combination of removing each of two constraints separately does not result in a vacuous requirement: either of the two relaxed versions of the requirements is acceptable, but the system must satisfy at least one of them.

The combination of a relaxing change and a constraining change selectively loosens and also tightens the requirements:

$$b[a \wedge b](a \wedge b \wedge c) = b \wedge (a \Rightarrow c)$$

The constraint b is common to all three versions, and it appears in the combination as well. The first change drops the constraint a, while the second change adds the constraint c. In the combination, the new constraint c must hold only in the cases where the original constraint a is satisfied. This moderation of the constraining change is due to the presence of the relaxing change; if we do not remove the constraint a then the new constraint c is added unconditionally:

$$(a \wedge b)[a \wedge b](a \wedge b \wedge c) = a \wedge b \wedge c$$

```
FUNCTION spell_1
  MESSAGE spell(report: sequence{word} ) REPLY(errors: sequence{word})
    WHERE ALL(w: word :: w IN errors <=> w IN report & ~(w IN dictionary))

  CONCEPT dictionary: set{word}
    -- The words in the Oxford English Dictionary.
END

INSTANCE word IMPORT Subtype FROM type
  WHERE Subtype(word, string),
    ALL(c: character, w: word :: c IN w => c IN ({a ..  z} U {A ..  Z}))
END
```

Figure 3: Specification of Initial Spelling Checker

To illustrate the effects of these rules in a more realistic context, consider the specification of a simple spelling checker whose base version is shown in Figure 3. We focus on the spell command. Figure 4 shows two changes to the behavior of this command, and the result of combining the changes using the method outlined in section 3.

All of the change merges in the examples follow directly from the definition, after simplification using the laws of ordinary propositional logic. These simplifications were

```
-- base version:
MESSAGE spell(report: sequence{word} ) REPLY(errors: sequence{word})
  WHERE ALL(w: word :: w IN errors <=>
            w IN report & ~(w IN dictionary) )

-- first modification:
MESSAGE spell(report: sequence{word} ) REPLY(errors: sequence{word})
  WHERE ALL(w: word :: w IN errors <=>
            w IN report & ~(w IN dictionary) & ~acronym(w) )

-- second modification:
MESSAGE spell(report: sequence{word} ) REPLY(errors: sequence{word})
  WHERE ALL(w: word :: w IN errors <=>
            w IN report & ~(w IN dictionary) ),
            sorted{less_or_equal@word}(errors)

-- result of change merging:
MESSAGE spell(report: sequence{word} ) REPLY(errors: sequence{word})
  WHERE ALL(w: word :: w IN errors <=>
            w IN report & ~(w IN dictionary) & ~acronym(w) ),
        sorted{less_or_equal@word}(errors) |
          ALL(w: word :: w IN report & ~(w IN dictionary) & acronym(w) &
              ~(w IN errors))
```

Figure 4: Merging Changes to the Spelling Checker

performed manually and then checked via an automatic simplifier for propositional logic that is implemented using term rewriting with respect to a canonical set of rewrite rules.

The base version has only the most basic requirements: there is only one dictionary, and there are no constraints on the order of the words in the output. The first enhancement introduces the modified requirement that acronyms (which contain only capital letters) are never reported as spelling errors. The second enhancement adds a requirement for sorting the output. The result of merging the two changes includes the acronym modification, but requires sorting only in the cases where the acronym modification did not take effect. This is a consequence of the minimal change principle [8] implicit in the change merging formula. In this case, a review by an analyst concludes that the case where the sorting requirement is suspended is impossible: the dictionary (a constant in the specification) cannot be empty in any acceptable version of a spell checking system, as would be required by the second condition in the last quantifier of Figure 4. In general, application of the change merging rules can highlight cases where requirements changes interact. These cases can then be reviewed by people to check whether a subtle interaction was missed or misjudged.

The implementation of the change merging definitions for specifications is straightforward, just as is the implementation of weakest preconditions for loop-free code. The

difficulty of automatic application lies in the simplification step in both cases: since most logics that are useful for specification are not decidable, it is in general impossible to do a perfect job of simplification. For these logics, there is no computable canonical form in which all tautologies reduce to the logical constant "true" and all contradictory statements reduce to the logical constant "false". In the above examples, simplification using only the propositional structure of the formulas was sufficient to get useful results, even though human judgement was needed to recognize constraints that hold in the problem domain, but are not universally true in the logical sense. However, even for decidable systems such as propositional logic, the existence of a canonical form does not solve the problem completely, because the result produced by the simplifier does not resemble the original formulas and is typically hard to read. Manual simplification was needed in the above examples to make the results readable by people. Heuristic methods that try to match the original structures as far as possible would be useful for practical decision support. This is an area for further research.

5 Conclusions

We have presented a method for merging changes to a black box software specification, particularly those expressed using logic. Since logic has a natural Boolean algebra structure, the application of standard change merging models was straightforward once the refinement oredering for the larger scale aspects of system specifications were determined. Although the definition of the Boolean difference operation for logical assertions is a direct consequence of this algebraic structure, it is an unfamiliar operation and its behavior is somewhat counter-intuitive. We found that the effects of the change merging formulas were hard to predict without performing the detailed calculations prescribed by our method.

The main issues remaining for practical application are verifying the conformance of these models to the actual intent of designers who wish to combine their changes, and providing effective automation support for assertion simplification that can put synthesized assertions into a form readily understood by people.

Our previous research has explored formal models of the chronological evolution history [21]. This model has been applied to automate configuration management and a variety of project management functions [1]. The ideas presented in this paper are a promising basis for improving these capabilities, particularly in the area of computer aid for extracting useful design rationale information from a record of the evolution of the system.

Challenges facing future research on meaning-altering changes are to span the software design space using a set of manageable changes with precise and expressive representations, to provide automatic procedures for suggesting applicable changes, and to construct automatic or computer-aided procedures for decomposing manual design changes into sequences of primitive changes. Successful research in this direction and its future applications will support software design automation with great scientific and economic impact.

References

[1] S. Badr, Luqi, Automation Support for Concurrent Software Engineering, *Proc. of the 6th International Conference Software Engineering and Knowledge Engineering*, Jurmala, Latvia, June 20-23, 1994, 46–53.

[2] F. Bauer et al., *The Munich Project CIP. Volume II: The Program Change System CIP-S*, Lecture Notes in Computer Science 292, Springer 1987.

[3] V. Berzins, On Merging Software Enhancements *Acta Informatica*, Vol. 23 No. 6, Nov 1986, pp. 607–619.

[4] V. Berzins, Luqi, An Introduction to the Specification Language Spec, *IEEE Software*, Vol. 7 No. 2, Mar 1990, pp. 74–84.

[5] V. Berzins, Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada*, Addison-Wesley Publishing Company, 1991, ISBN 0–201–08004–4.

[6] V. Berzins, Software Merge: Models and Methods, *Journal of Systems Integration*, Vol. 1, No. 2, pp. 121–141 Aug 1991.

[7] D. Dampier, Luqi, V. Berzins, Automated Merging of Software Prototypes, *Journal of Systems Integration*, Vol. 4, No. 1, February, 1994, pp. 33–49.

[8] V. Berzins, Software Merge: Semantics of Combining Changes to Programs, *ACM TOPLAS*, Vol. 16, No. 6, Nov. 1994, 1875–1903.

[9] V. Berzins, *Software Merging and Slicing*, IEEE Computer Society Press Tutorial, 1995, ISBN 0-8186-6792-3.

[10] V. Berzins, D. Dampier, Software Merge: Combining Changes to Decompositions, *Journal of Systems Integration*, special issue on CAPS (Vol. 6, No. 1–2, March 1996), pp. 135-150.

[11] V. Berzins, Recombining Changes to Software Specifications, *Journal of Systems and Software*, to appear, Aug, 1998.

[12] M. Feather, A System for Assisting Program Change, *ACM Transactions on Programming Languages and Systems*, Vol. 4 No. 1, Jan 1982, pp. 1–20.

[13] M. Feather, A Survey and Classification of some Program Change Approaches and Techniques, in *Program Specification and Change (Proceedings of the IFIP TC2/WG 2.1 Working Conference)*, L.G.L.T. Meertens, Ed., North-Holland, 1987, pp. 165–195.

[14] M. Feather, Constructing Specifications by Combining Parallel Elaborations, *IEEE Transactions on Software Engineering*, Vol. 15 No. 2, Feb 1989, pp. 198–208.

[15] S. Fickas, Automating the Transformational Development of Software, *IEEE Transactions on Software Engineering*, Vol. 11 No. 11, Nov 1985, pp. 1268–1277.

[16] W. Johnson, M. Feather, Building an Evolution Change Library, *12th International Conference on Software Engineering*, 1990, pp. 238–248.

[17] E. Kant, On the Efficient Synthesis of Efficient Programs, *Artificial Intelligence*, Vol. 20 No. 3, May 1983, pp. 253–36. Also appears in [26], pp. 157–183.

[18] Luqi, M. Ketabchi, A Computer Aided Prototyping System, *IEEE Software*, Vol. 5 No. 2, Mar 1988, pp. 66–72.

[19] Luqi, V. Berzins, R. Yeh, A Prototyping Language for Real-Time Software, *IEEE Transactions on Software Engineering*, Vol. 14 No. 10, Oct 1988, pp. 1409–1423.

[20] Luqi, Software Evolution via Rapid Prototyping, *IEEE Computer*, Vol. 22, No. 5, May 1989, pp. 13–25.

[21] Luqi, A Graph Model for Software Evolution, *IEEE Transactions on Software Engineering*, Vol. 16, No. 8, pp. 917–927, Aug. 1990.

[22] Luqi, Specifications in Software Prototyping, Proc. SEKE 96, Lake Tahoe, NV, June 10-12, 1996, pp. 189-197.

[23] R. Lutz, Analyzing Software Requirements: Errors in Safety-Critical Embedded Systems, TR 92-27, Iowa State University, AUG 1992.

[24] D. Parnas, P. Clemens, A Rational Design Process: How and Why to Fake It, *IEEE Transactions on Software Engineering*, Vol. 12 No. 2, Feb 1986, pp. 251–257.

[25] S. Horowitz, J. Prins, T. Reps, Integrating Non-Interfering Versions of Programs, *ACM Transactions on Programming Languages and Systems*, Vol. 11 No. 3, Jul 1989, pp. 345–387.

[26] C. Rich, R. Waters, Eds., *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[27] D. Smith, G. Kotik, S. Westfold, Research on Knowledge-Based Software Environments at Kestrel Institute, *IEEE Transactions on Software Engineering*, Vol. 11 No. 11, Nov 1985, pp. 1278–1295.

[28] W. Swartout, R. Balzer, On the Inevitable intertwining of Specification and implementation, *Communication of the ACM*, Vol. 25 No. 7, July 1982, pp. 438–440. Also appears in *Software Specification techniques*, N. Gehani, A.D. McGettrick, Eds., 1986, pp. 41–45.

Domains as a Prerequisite for Requirements and Software

Domain Perspectives & Facets, Requirements Aspects and Software Views

Dines Bjørner

Bernried, 12 October 1997 — Lyngby 5 March 1998

Abstract

We take software [systems] engineering to consist of three major phases: domain engineering, requirements engineering and software [systems] design engineering. We outline these and emphasise domain perspectives andfacets, requirements aspects and software architecture and program organisation views.

This paper is the direct result of a US Office of Army Research October 12–14, 1997 workshop on *Requirements Targeting Software and Systems Engineering* held at Bernried am Staarnberger See, Bavaria, Germany. In consonance with the aims & objectives of that workshop we conclude some subsections with a set of meta–requirements (i.e. requirements to *software engineering, its research, education and practice*).

The paper is discursive and informal: we identify a number of methodological principles, techniques and tools. Not all such (hence discursive) and not all necessarily formalisable (hence informal). Wrt. the latter: one cannot formalise the principles that are needed in a systematic, well-guided process of selecting and applying techniques and tools in the analysis and synthesis of specifications — whether of domain, requirements or software. Instead we are left to conjecture the usefulness of certain such principles, techniques and tools. Sometimes such conjectures are refuted when better principles, techniques and tools are proposed.

Some sociological issues of 'formal methods' are summarised (in section 4.5).

Since this paper will appear in a workshop proceedings with a number of other papers from that workshop, the paper will not repeat the relevant points made by other workshop participants and supposedly published in their contributions. I refer, amongst several, to contributions made at the workshop by Carl Gunther, Anthony Finkelstein, Michael Jackson, Tom Maibaum and others.

On issues of requirements, I have, in particular, benefited much from [100, 99, 142]. The handy book [99] is simply a pearl: delightful and thought provoking!

Contents

1 Introduction

We present an interpretation of the state of affairs, i.e. level of professionalism, in software engineering, the thesis of this paper, and a first justification of the thesis.

1.1 The State of Affairs

In the US more than US $ 180 billion was spent in 1996 on software development projects that were curtailed, given up and abandoned, because management did not believe they could conclude these projects. Most often cited reasons for this failure were: The requirements were insufficient, elusive or changed, and the domains were not properly understood. This is according to the article: *Formal Methods: Promises and Problems*, IEEE Software, Vol. 14, No. 1, Jan. 1997, pp. 73-85. US $ 180 billion is a sizable amount.

To this we can add that even if and when developers get the domains and requirements right, they often get the implementation wrong — but this aspects apparently does not bother people who willingly buy MicroSoft software even when it is known that it definitely contains thousands upon thousands of (even known) bugs.

A survey, in Europe, by the industry 'Think Tank' ESI, the European Software Institute in Bilbao, Spain, records that most software industry managers express that their must urgent problems have to do with grossly insufficient methods, techniques and especially tools for coping with requirements definitions — and they are basically unaware of the pre-cursor to requirements develeopment, namely domain modeling!

1.2 The Thesis — and the Contributions

The thesis of this paper is that the kind of domain, requirements, software architecture and program organisation principles and techniques expounded in this paper seems to offer workable solutions to the problems. At least they address the issues "head-on" and in a systematic manner not yet reported this extensively. Besides the 'triptych' decomposition of development into domain, requirements, software architecture and program organisation organsiation, we also would offer the identification of domain perspectives and facets, the requirements aspects and the redefinition of software views as a contribution.

1.3 The 'Triptych'

Software (systems) engineering, to us:

- "Starts"[1] with domain engineering. Result: A formal theory $\mathcal{D}$.
- "Goes on" with requirements engineering. Result: A formal theory $\mathcal{R}$.
- "Concludes" with software (systems) design engineering. Result: A formal theory $\mathcal{S}$

We expect the following kind of relationship[2] to hold between $\mathcal{D}, \mathcal{R}, \mathcal{S}$:

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

A classical example is compiler development:

- **Domain:**

  We define the concrete (BNF) and abstract syntaxes the abstract mathematical semantics of the source and target languages (programming, resp. (e.g.) machine), and a proof system for the source language.

- **Requirements:**

  We define the specific compiler and run-time requirements: fast compilation, or fast execution, or extensive compilation diagnostics, or extensive run-time execution diagnostics, or some combination of these (+ many other facets).

[1] We often put double quotes around words when their meaning is only approximate. The actual sequence from 'start' to 'conclusion' is usually iterative and "spiral"!

[2] See Carl Gunther [81].

- **Design:**

  We develop the compiler and run-time systems: possibly a single or a multi-pass compiler while using such design tools as lexical scanner generators, (possibly syntax error correcting) parser generators, (possibly a variety of) attribute grammar evaluators, etc.

We define:

- A **method** is a set of principles, techniques and tools for analysing problems and for selecting and applying techniques and tools in order efficiently to construct an efficient artifact — here software (a systew).
- **Methodology** is the study and knowledge of methods.

Jackson [100, 99] has proposed a decomposition of the unending variety of problems for which software engineers are expected to develop software solutions, into a possibly in[de]finite set of problem frames. Our compiler development example above thus is archetypical of a 'translation problem frame'. Each frame is characterised by its distinct cluster of development principles, techniques and tools. Therefore we speak not of one method but of possibly an in[de]finite collection of methods.

Common to these, we argue [36], is that they all evolve along domain, requirements and design engineering axes.

1.4 First Justifications

Software offers functions. Usually the client expresses expectations about the software to be delivered: that is, requirements that include characterisations of externally observable properties of these functions.

So before we develop software we ought know very precisely the externally observable, that is: the user expectations about the concepts & facilities to be offered by the software.

These requirements usually deal with components, actions and behaviours of the client domain, the application domain. And usually the requirements are expressed in terms of terms (nouns and verbs) that 'reside' in — are special, professional terms of — the domain.

So before we develop the requirements definition it seems a good idea to recognise, discover and capture the domain and to make precise the structure and meaning of all the special, professional domain terms otherwise used in informally expressing requirements.

The situation is not new. In other engineering branches we encounter the need for securing the domain understanding — and usually also formally — before requirements are expressed. In control engineering for aerospace the domain is typically that of Kepler's and Newton's laws. So there was Johannes and Isaac working on their laws only to get an understanding of celestial mechanics, or to prove the existence (or non-existence) of God, or at least to be able to calculate (predict) planetary movements. There was, in those days, little expectation of the laws being generally applicable, and certainly not to for example automotive engineering or satellite orbit determination. Usually a problem in these areas starts with the control engineer specialising the normative theories of Kepler and Newton to the specifics of the problem at hand. Requirements are then usually expressed as constraints on the mechanical behaviours specified (typically) by the differential equations that describe the instantiated problem domain (i.e. instantiated theory). Control design then finds controllers ($\mathcal{S}$) and show that they satisfy the requirements ($\mathcal{R}$) under theassumption of the domain model ($\mathcal{D}$). One

would never dream of hiring a person to develop control for flight systems unless they were well-educated and professionally specialised in the appropriate control, respectively aerospace engineering disciplines.

Similarly for electrical and communications engineers, etcetera. They, or you, may not think of the laws of electronics (Ohm's, Kirschoff's, etc.), or Maxwell's equations, constitute domain models; but that is what they do! And so it goes. One would never dream of hiring a person to develop sensor electronics or space communications for flight systems unless they were well-educated and professionally specialised in the appropriate electronics and communications engineering disciplines.

It is high time that software engineers become as smart and mature, productive and responsible, professional and specialised as other engineers. On one hand most practising software engineers today are unaware of the advances wrt., and the broad applicability of formal techniques — available for many years now. On the other hand: how can they believe that they can develop software for banking, railway or manufacturing industry applications without having studied — or themselves developed — appropriate domain theories for these application areas.

2 Domain Engineering

The aim of domain enginering is to develop, together with stake-holders of, or in, the selected domain, a precise set of concordant descriptions of the domain, a set that the parties, developers and stake-holders, can agree upon.

Thus we foresee some set of loose contractual obligations binding the two parties.

From a formal point of view, domain engineering establishes the theory $\mathcal{D}$.

2.1 Example Domains — Infrastructures

Examples of domains, limited to infrastructure systems, are:

- **Transport Systems:** [119]
  - **Railways** [16, 37, 47, 62, 22, 23]
  - **Air: Trafic and Lines** [19, 12]
  - **Metropolitan Transport** (bus, train, taxi, ec.) [140, 59]
  - **Shipping**
- **Manufacturing Industry:** [74, 75, 6, 102, 7, 101, 103]

  Infrastructures 'connecting' software packages(across each of the individual (intra: (i)), respectively between these (inter: (ii)) spectra: (i) marketing, design, order processing, shop floor production, wharehousing, sales, service, etc., and (ii) suppliers and consumers, producers and traders, etc.
- **Financial Service Industry:**

  With individual models, and with models that span across the entire industry:
  - **Banking:** Demand/deposit, savings & loan, investment, etc. [20, 107, 21]

- **Insurance Companies:** Health, life, accident, theft, risk, etc.
- **Securities:** Exchanges, brokers, traders, etc.
- Etcetera: credit card and bank cheque clearing; portfolio management, etc.

- **Ministry of Finance:** [57]
- **Health Care Systems**
- **Decision Support Systems for Sustainable Development:** [85, 17]
- **Resource Management:** [18, 123]
- *&c.*

2.2 First Aims of Domain Engineering

An aim of domain engineering is to provide partial and sufficient answers to questions like: *What is a railway company?, a transport industry?, a bank?, a financial service industry?, a manufacturing enterprise?,* respectively: *What is a manufacturing industry?* The question is expected answered without any reference to possible requirements to potential software, and (certainly) without any reference to implementations of such software — as the domain in question already exists, existed (or still exists) without any software for a long time. We expect the domain description to be both informal, but in the professional language of the domain (including its dagrammatic and other linguistic devices), and formal, in the form of a formal specification of the crucial terms (viz.: nouns and verbs) of the professional domain language(s). We will not in this paper analyse any perceived problem of extracting or communicating this domain knowledge.

A domain description is a model of the domain, that is: necessarily an abstraction. To conquer possible complexities of the domain we may focus on various **perspectives** of the domain — i.e. understandings as held by various groups of domain stake-holders.

2.3 Domain Models — An Example: Railway Systems

Let us try give an example. The problem of giving an example of a domain model of a sizable domain is that the example need be kept within limits, but the domain is usually perceived, by the reader, as being inordinately complex and "large".

First we give an informal description — which ideally consists of a triple: a **synopsis**, a **narrative** and a **terminology**. Then we present a formal specification (in the RAISE [77] Specification Language: RSL [76]).

2.3.1 Synopsis, Narrative & Terminology

- **Synopsis:**[3]

  A synopsis is a terse informal text which — by mentioning, in a reasonably structured and "softly" enumerative way, the names of a number of components, actions and behaviours — may lead the reader onto *'what the whole thing is all about'*.

[3] The synopsis shown may be claimed to be sufficient.

The domain (and hence its description) is normative,[4] *and is that of railway systems. Common to these we find that railway systems can be characterised in terms of the railway net with lines and stations, of timetables, of traffic (in terms of the movement of trains), of passenger and freight ticket and space reservation, ticketing, loading and unloading, of the shunting and marshalling of trains, of various concerns about rolling stock, of the management and human operation of the system, and of repair, maintenance and development of all of these resources.*

- **Narrative:**[5] A narrative is a careful description of all the relevant notions of the described "thing":
  - Railway Nets: *A net consists of lines and stations. A line connects two distinct stations.*

    Stations consists of tracks and other combinations of rail units. Lines and tracks consists of sequences of linear units.[6] *Lines can be seen as a sequence of blocks which are then sequences of linear units.*

    Units have connectors. A linear unit is a unit with exactly two connectors. A switch (unit) has three connectors. A cross-over (unit) has four connectors. Etc.[7]

    Connectors have identity. At most two units can share at most two connectors and do so if they have identical identity.[8]

    With a unit we can associate the set of its potential states. A state of a unit is the set of open paths through a unit. A path of a unit is a direction through the unit in which traffic (train movement) is possible. Over time a unit can undergo state changes.[9]
  - Timetables:

    Several notions of timetables may (co-)exist:

    A timetable may be very basic and show, for example for each train number, the route of the train, that is: the sequence of station visits together with the train arrival and departure times at those stations. Or a timetable may additionally be train-dispatch oriented and may furthermore show train clearance and station routing information as well as approximately when (at which times) the train should be at which blocks along the lines. Or a timetable may be passenger-oriented and also show quantity and quality of seats and sleepers. Etcetera.

    Stations have unique names.
  - Traffic:

[4] By a normative description we mean a description of a class, rather than a particular member of the class.

[5] The narrative shown here is much too simplified — but the example shows what is meant by a narrative.

[6] Pragmatically: Example tracks are: passenger and freight train through tracks, passenger train platform tracks, freight train sidelines, load/unloading tracks and shunting and marshalling yard tracks.

[7] Pragmatically: Connectors seem to be an artificial "device" needed in order to easily define nets. Units are similarly pragmatically chosen atomic quantities.

[8] Pragmatically: the two units are connected at the "join" of those two identically identified connectors.

[9] Pragmatics: The state of a unit may be effected by the setting of switches and signals — but so far we abstract that. The state of a unit serves to route trains properly. Trains are intended to only pass in the direction of an open unit: from one connector towards another. Whether trains obey the state setting is a matter outside the domain. In the domain we must also model human errors, technology failures and catastrophes.

Traffic is the continuous movement — interspearsed by temporary stops — of trains along the net. Trains have both train numbers and unique train identifiers.[10]

By movement we mean that if at two relatively close times (say separated by a few seconds, t, t'') a train is moving snd is at, i.e. occupy, two distinct sequences of units, i.e. at positions (p, p''), then at any time (t') in-between, the train is at monotonic positions (p') in-between p and p''.[11]

Traffic can be observed, and ideally, as the above 'continuous' function, or traffic can be scheduled (planned). Scheduled traffic may be in the form of a discretised prescription, as in the train-dispatch oriented timetable.

– &c.

- **Terminology:**[12] A terminology is an alphabetically sorted list of concise, informal definitions.

  We only examplify a few terms.

  Capitalised terms used in definitions refer to separate entries. Defined terms are listed alphabetically.

  – Connector: *A Connector has an Identity and is further undefined. At most two Units may share at most equal identity Connectors.*

  – Hump: *A Hump is a Unit and is a notion of Marshalling.*[13]

  – Incoming (Marshalling) Tracks: *A set of one or more Tracks form an Incoming (Marshalling) Track configuration if the Tracks at one end are 'fanned-in' (merged) into a Hump.*[14]

  – Line: *A Line is a non–empty sequence of Linear Units. A Line 'connects' two Stations.*

  – Marshalling: *Marshalling are the actions of decomposing a (potentially unending) series of sequences of freight cars, passenger waggons, etc., into a potentially unending series of set of (parallel) sequences of freight cars, passenger waggons, etc.*

  – Marshalling Yard: *A Marshalling Yard consists of three main parts: a small set of one or more Incoming Tracks (otherwise connected, at an incoming end, to (other) units of a Station), a usually large set of Outgoing Tracks (otherwise connected, at an outgoing end, to (other) units of a Station), and a Hump. Usually routes through the Marshalling Yard are only possible from the Incoming to the Outgoing Tracks.*

[10] Pragmatics: Two or more trains on the net may have identifical train numbers — since their journey may last longer than the time interval by means of which a timetable may be defined. In this case we may wish to use train identities in order to be able to distinguish any two trains.

[11] By monotonic movement we mean that the direction of the train does not change in the closed interval $[p, p'']$.

[12] Another term could be: 'Dictionary'. This one is very "sparse", but we hope sufficient for the reader to get the idea.

[13] Pragmatics: A Hump 'connects' Incoming and Outgoing Tracks and permit the orderly selection of cars, waggons, etc. from Incoming Tracks and their distribution to appropriate Outgoing Tracks.

[14] Pragmatics: A series of incoming sequences of cars and waggons may be routed onto the other end so that individual cars or waggons may be routed onto the Hump from either Incoming Track.

- Net: *A Net consists of a set of one or more (known) Lines and a set of two or more (known) Stations. All known Lines must connect known Stations. Each known Station must be connected to at least one known Line.*
- Outgoing (Marshalling) Tracks: *A set of one or more Tracks form an Outgoing (Marshalling) Track configuration if from a Hump there is a 'fan-out' to the Tracks at one end which at the the other end are connected to other Station Units.*[15]
- Open Path: *An Open Path is a Path which is in the current Traffic State of a Unit.*
- Open Route: *An Open Route is a Route all of whose Paths are in the current Traffic State of the Net.*
- Path: *A Path is a way through (a direction along) a Unit.*
- Route: *A Route is an orderly connected sequence of Paths.*
- Station: *A Station consists of a set of one or more Tracks and a set of Units.*[16]
- Traffic State of a Net: *The totality of the Traffic States of the Units of a Net makes up the Traffic State of a Net.*
- Traffic State of a Unit: *A Unit can, at various times, "occupy" one or another Traffic State.*[17]
- Track: *A Track is a linear sequence of one or more Linear Units.*[18]
- Unit: *A Unit is — for the purposes of this description — a smallest 'unit' of rail. Units serve to compose Nets. Nets can be decomposed into Units. There are Linear Units, Switch Units, Crossover Units, Turntable Units, Track End Units, etc.*[19]

2.3.2 Formal Specification

- **Nets:**

```
type
  Net, Lin, Sta, Trk, Uni, Con
value
  obs_Lins: Net → Lin-set
  obs_Stas: Net → Sta-set
  obs_Unis: (Net|Sta|Lin) → Uni-set
  obs−Cons: Uni-set
  LinStas: Lin → Sta × Sta
axiom
```

[15] Pragmatics: It may be better to say that a Hump is the root of a fan–out to a number of tracks, where the fan-out is a configuration of mainly switch units whose "leaves" are connected to one end of the Outgoing Tracks.

[16] Pragmatics: The Tracks serve a main purpose of a Station: to Load and Unload Passengers and Freight, to temporarily 'park' trains, to Marshal a set of Trains into another set, and to let Through Trains pass the Station. The (other) Station Units serve to Route Trains between Lines and Tracks.

[17] Pragmatics: A Traffic State of a Unit indicates a number of Paths through that Unit as being open for Traffic.

[18] Pragmatics: Tracks can be classified to belong to one or more of either: Platform Tracks, Through Tracks, Shunting/Sideline Tracks, Freight Load/Unloading Tracks, Incoming and Outgoing Marshalling Tracks, etc.

[19] Pragmatics: You may wish to think of a linear Unit as a pair of rails, a large set of sleepers, each sleeper fastened to the rails by 'nails', etc.

∀ n:Net •
  let stas = obs_Stas(n),
    lins = obs_Lins(n) in
  ∀ s,s′ in stas s≠s′ ⇒ ∃ l in lins • LinStas(l) = (s,s′)
  end
...

- **Timetable:**

type
  TT, T, Sn, Tn
  TimeTable = Tn $\overrightarrow{m}$ (Sn $\overrightarrow{m}$ (T × T))
value
  obs_Sn: S → Sn
axiom
  /* observed station visits are linearly ordered: trains */
  /* arrive at stations before they leave, and arrival times */
  /* at 'later, subsequent' stations follow departure */
  /* times from 'earlier, previous' stations. */

The times shown in one way (TimeTable) of observing (i.e. projecting) an abstract timetable (TT) are modulo some reasonable interval, say working days, week-ends, holidays.

- **Traffic:**

type
  TF, Tid, Rou
  rTraffic = T $\xrightarrow{\sim}$ (Tid $\overrightarrow{m}$ Rou)
  sTraffic = T $\overrightarrow{m}$ (Tid $\overrightarrow{m}$ Uni)
value
  obs_Tn: Tid → Tn
  obs_UniRou: Rou → Uni*
  obs_Traffic: TF → rTraffic

- **Managed Nets and Traffic:**

type
  MN, MTF
  MgdNet = T → Net
  MgdTraffic = T → (Net × ((Tid $\overrightarrow{m}$ Rou)×(Tid $\overrightarrow{m}$ Uni)))
axiom
  /* Routes and Units of train positions must be thos of the */
  /* net. Unit positions of scheduled traffic must have */
  /* appropriately open paths. Etcetera. */

A managed net reflects the time changing set of unites and states of units. A managed traffic reflects the managed net, the real and the scheduled traffic.

2.4 Domain Perspectives

The concept of perspective is a pragmatic one. It serves to decompose an otherwise large domain description into a more manageable, structured set of related descriptions — each corresponding, as closely as possible, to a stake-holder perspective. The pragmatics, at the domain level of perspectives is that each perspective, i.e. each sub-description covers a distinguishable set of closely related components, properties, actions and behaviours of the domain being described.

- **Domain Perspective:**

  We can (formally) define a **perspective** as a partial specification of a domain, consisting of a type space and a set of functions.

We continue the railway systems example from above.

- **Base Perspectives:**

  It seems that railway nets and timetables form the main two base perspectives.

  As a minimum any stake-holder seems to agree that the railway net in terms just of lines and stations and a simple observation of timetables suffice to characterise many aspects of railway systems.

  The timetable and the net are related by stations and — implicitly — by lines.

- **Signalling Perspectives:**

  By signalling, at the intrinsic level (see domain facets, section 2.5) of domain descriptions, we mean just the state of the net (including its units). Signalling is a control perspective.

  For trains to actually journey across the net through stations and along lines signalling must be in effect: paths and routes must be opened and closed in order to ensure safe and speedy traffic.

  So we need to further detail the net into units and their states, open and closed. Managed nets and traffic may be a way to describe signalling.

- **Passenger Perspectives:**

  Passengers, in addition to the basic net and timetable descriptions, as well as railway system staff with whom they interact, have a perspective of the railway systems as somehow embodying ticket reservation, cancellation, etc. Passenger perspectives are user perspectives.

```
type
  B, Date, Tn
  Occ = Tid ~m~> ((Sn × Sn) ~m~> (Free × Bound))
value
```

ide: Date × Tn $\xrightarrow{\sim}$ Tid
res: Date × Tn × (Sn × Sn) × B $\xrightarrow{\sim}$ (B × Ticket)
can: Ticket × B $\xrightarrow{\sim}$ B
axiom
/* Can only reserve if free seats. Accepted reservation*/
/* leads to less free sests, more booked ones */
/* Cancellation only if ticket is validly reserved. */
/* Etcetera. */

We should have stressed before, and will here stress, that domain descriptions of components, like B, and actions, like reservations and cancellations, and constraints, like the axioms stated, are abstractions of "real world"[20] phenomena which may not (or may already) be machine supported. That is: our descriptions are assumptions about the "real world", with or without computing support for what is being described. In other words, B, may or may not become the basis for a computerisation, etc.

There are many other (rail net and rail service development (i.e. plant), statistics, timetable planning (i.e. management), etc.) perspectives. It is not the purpose of this paper to enumerate as many as possible, nor to further analyse the concept of perspectives.

2.5 Domain Facets

The 'domain perspective' concept was application-oriented. Each perspective portrayed a suitably and pragmatically chosen part of the domain.

The 'domain facet' concept is a somewhat technical one, but still is basically determined on pragmatic grounds.

Any domain has some intrinsic parts. These are parts which reflect 'stable' properties of the domain, that is: properties which remain properties also when the "hard" technologies that 'support' the domain change, or when the rules & regulations that 'govern' stake-holder domain actions and behaviours change, etc.

- **Domain Facet:**

  We can (formally) define a **facet** as a partial specification of a domain, consisting of a type space and a set of functions.

In the following we will illustrate 'clusters' of these facets in the context of the railways example:

- **Intrinsic Facets:**

  By an intrinsics of a domain we loosely (pragmatically) mean those facets which remain invariant under changing support technologies, changing enterprise system or infrastructure rules & regulations, changing stake-holder behaviours, etc. That is: We define intrinsics as a core and relative to (modulo) other facets.

  Therefore an improved characterisation of intrinsics should emerge as we next deal with supposedly non-intrinsic facets.

[20] No-one knows what 'real traffic is. Therefore we put double (tongue-in-cheek) quotes around that concept.

- **Support Technology Facets:**

  Much of the technology that reside in the railway domain is changing regularly while its intrinsics remain stable.

  - Switch Technology:

    One example is the switch technology. In the early years of railways switches were thrown manually, by railway staff, one per switch! Later mechanical gadgets, including momentum amplifiers, were connected by thick wires to a central cabin house which predominantly featured a row of 'throwers'. Now we find that combinations of switches are activated electronically and electrically through so--called solid-state switches. The same underlying, intrinsic concept, a switch, has its internal functioning determined by varieties of support technologies.

  - Signal Technology:

    Another example is, or was, the visible, mechanical signals consisting of a tall mast (or pole) to which are affixed, at the top, for example one or two 'flags'. These are hoisted or lowered through cabin-located 'throwers'. Later some such mechanical signals were replaced by signals consisting of not so tall poles on which are fixed red/yellow/green or just red/green lamps. In future we can foresee that all such signals are replaced by radio messages sent to each individual train informing it of whether to make a stop or not, including actually performing that control — the meaning of a signal.

  - Sensor Technology:

    Yet a third example is the following. It is based in how we observe traffic. In the intrinsics we claim that traffic is a continuous function from time (at least within a suitably chosen interval) to train positions. In "physical reality" we know that at whichever time we choose to observe the traffic there will indeed be trains. In the "observable reality" we cannot observe all the time all the positions. Instead we place observers at suitably chosen points (units). That is, wrt. **space**, *we choose to sample, and this spatial sampling discretises our observations. Also: we do not observe all the* **time**, *but chooses to let the observers inform us only of changes: now there is a train starting to pass by in that direction, now the trains ends passing by. That is: rather than being subject to continuous evaluation we discretise in the form of observable events. The observers form a kind of 'support technology'. In the "old days" the observers were railway staff that might have used some form of telegraphic or telephonic equipment to inform a more-or-less central gathering of observations. Today optical sensors (optical gates) may be used as observers (and perhaps with extended functionality). The point is: the support technology changes.*

  - *&c.*

(The point is also that) Support technology may fail. In the intrinsics observations, switch setting, unit openings and closing were "ideal". In the presence of possibly and probabilistically failing technology switches may fail to change state, signals may "get stuck", and sensors may register a 'train-passing-by' event when in "real reality" there is no such train, or vice-versa: may fail to register a passing train.

- **Modeling:**

  The intrinsics descriptions (models, whether informal or formal) must therefore be extended (enriched) to include the components, actions and behaviours of support technology.

  Typically the models must incorporate real-time, safety criticality, (failure) probabilistic, etc. properties.

  Formal specification languages that are able to cope with some of these facets are the Duration Calculi of Zhou Chaochen [45, 50, 49, 43, 44, 92, 48, 93].

- **Rules & Regulations Facets:**

  Written procedural guidelines exists in most man-made domains. They are intended to regulate the actions and behaviours of staff in operating, i.e. interacting with the domain.

  - **Railways:**

    Examples, again relating to the railways, are:

    * Trains at Stations:
      In China, probably due to some pretty disastrous train crashes at stations, there is a rule, concerning acceptance of incoming trains and dispatch of departing trains at stations. This rule states that in any n minute interval (where $n = 5$) there must at most be one train arriving or departing a station — even though some stations may have ample tracks and disjoint routes from and to lines: sufficient to actually receive or send several trains simultaneously.[21] But a rule is a rule!
    * Trains along Lines:
      Lines may be decomposed into blocks, with blocks delineated by for example signals. The purpose of blocks is usually to ensure that there is at most one train in each block, or that there is at least one block between any two trains on the same line. Again blocking may be introduced in order to make it simpler to monitor and "control"[22] traffic along lines in order to ensure safety (no crashes). Thus some support technology (e.g. signals) may be a means to ensure a rule.
    * Dispatch & Rescheduling Rules:
      Rules giverning the dispatch and rescheduling priorities among train types (international vs. local passenger trains vs. similar grades of freight trains vs. military trains) abound.[23]
    * *&c.*

[21] This kind of rule is similar to air traffic at airports: Pairwise adjacent landings on any given runway must be separated by, say, at least 2 minutes. Similar for take offs. And any adjacent pair of a landing and a take-off, or a take-off and a landing must be separated by, say, 5 minutes!

[22] What exactly is meant by 'control' is left undefined.

[23] Especially these rules are changing rapidly these years in the light of the "ownership" decomposition of of railway systems: base net & signalling infrastructure in terms of one operator vs. commercial passenger and freight traffic in terms of possibly several, competing operators. They are changing in order to further competition.

– **Banks:**

Customer deposit monies into savings accounts, for example as 'exchanged' with a bank teller, involve 'interpretation', by the teller, of *rules & regulations* for posting such deposits. Depending on the customer *account contract* (in which the *rules & regulations* concerning all transaction are 'defined'), the clerk performs one or another set of actions (a 'script') "against" the account (i.e. banking) system. The *account contract* (generally set up when the account is first established) 'binds' concepts (i.e. concept identifiers) such as for fees, interest rates, loan limits, etc. to actual values. (This binding is reminiscent of environments *ENV* when modelling block-structured programming languages.) The domain model of deposit and other transactions are therefore modelled as a Bank Programming Language script. A script has two formal parameter lists, a transaction arument list and a contract identifier list. When performing a transaction, i.e. when invoking the script, transaction parameter values are bound to identifiers of the transaction argument list, while the (latest) contract environment is used to find the values of the contract parameter list.

Model:

```
type
  Pid, Cid, Cmd
  Bank = {accounts} m→ (C m→ Acc) ...
  ENV = Cid m→ VAL
  Script = (Pid* × Cid*) × Cmd
  Sn = {...,deposit,withdraw,save,borrow,...}
  Acc = ...
      ∪ {balance} m→ VAL
      ∪ {limit} m→ VAL
      ∪ {interest} m→ VAL
      ∪ {fee} m→ VAL
      ∪ {yield} m→ VAL
      ∪ {overdraw} m→ VAL
        ...
      ∪ {scripts} m→ (Sn m→ Script)
        ...
      ∪ {contract} m→ Text
      ∪ {env} m→ (ENV × T)*
        ...
Trans = Sn × VAL*

value
  int_Rou: C × Trans → Bank → Bank
  int_Rou(c,sn,vall)(b) ≡
    let a = (b(accounts))(c) in
    let ((pl,cl),cmd) = (a(c))(sn),
        env = (a(c))(env) in
    /* assert: */ len vall = len pl /* end assert */
```

```
let ρ = [ pl[i] ↦ vall[i] | i:Nat • i ∈ inds ]
      ∪ [ cl[i] ↦ env(cl[i]) ] in
int_Cmd(cmd)ρ end end end
```

Comments: Many other domains have rules & regulations that must be interpreted by humans, and the same rule & regulation may have to be interpreted according to some 'context'.

More generally on modelling we can say:

- **General Comments on Modeling:**

  The intrinsics and technology support descriptions (models, whether informal or formal) must therefore be extended (enriched) to include the components, actions and behaviours of *rules & regulations*.

  Procedural (human ↔ domain) matters tend to express logical properties for which also "exotic" logics like [auto]epistemic, belief, defeasible, deontic, and modal logics in general, may well serve as a basis for formalisation [66, 65, 67, 68].

  In general *rules & regulations* seem to be best modeled in terms of a special script language of commands. The command language is "tailored" to be able to access the domain state components. So: on one hand we do define major aspects of the intrinsics (basis), support technology, rules & regulations, human (stake-holder) behaviour, etc., using one (or another) specification language. But when it comes to typically the rules & regulations facet we defer further modelling to scripts written in a further defined domain specific (rules & regulations) script language. Now each such rule & regulation is then, in the domain model, associated with some script. Which script some rule & regulation is 'paired' with we do not model! But we should give example of sample interpretations of rules & regulations in terms of such rules!

  We refer to the item on Ground Staff Sub–facets page 17, in particular the continued bank example (page 17), for further on how humans may interpret rules & regulations.

- **Stake-holder Facets:**

  An important facet of the domain is the stake-holder concept: the staff of the system of interest within the domain (owners, managers, workers), the clients and customers, politicians, otherwise affected citizens, etc. Each have their own 'agenda' vis-a-vis the system, the domain in which it is embedded and more loosely connected issues which we may otherwise think of as "outside the domain".

  They express opinions, they have goals and objectives (not to be confused with requirements [to computing]), they manage other staff and other resources of the system (i.e. the enterprise, viz. a specific railway system operator), they operate its resources, they use the services or acquire the products of the enterprise, and they are otherwise "interested" in the well-being of the domain and its surroundings.

  Again some examples may serve to illustrate the points being made here:

- Owner Sub–facets:

  Owners of a system — an enterprise — residing within the system or domain may think of that system (or enterprise) in terms of goals and objectives that do not (later) easily translate into software requirements. Their facet is that of profitability, of growth, and of market share. Further subsidiary goals may then have to do with customer and staff satisfaction, with environmental (bio-sphere) concerns, etc. A model facet may try to cover this — but formalisation is probably difficult. It is impossible if there is no other formalisation of the domain, that is: formalisation of owner sub-facets may be enhanced in the presense of formal models of the domain. The system (state) 'variables' or 'indicators' in terms of which their sub-facets are to be formulated need be rather directly relatable to the domain model notion of state (and other) components.
- Manager Sub–facets:

  Managers acquire and dispose (i), allocate and schedule (ii), and deploy (iii) resources in order to meet goals and objectives at various levels: strategic (i), tactical (ii) and operational (iii) — respectively. At the higher, the strategic to tactical levels, one may be able to identify the kinds of components — including clients — involved and the kind (i.e. the type) of predicates that express satisfaction of goals and objectives — where the type of components are the type of the various resources being managed: time, people, equipment, monies, etc. Similar the decisions taken by management can be characterised, if neither algorithmically nor logically, then at least through their (algebraic) signatures. Report [18] shows that one can formally capture the domain sub-facets of the strategic, tactical and operational management of resources.
- Staff Sub–facets:

  The staff are the persons, "on the ground", being managed, and most directly exposed to the daily operations of the domain. They are the ones who directly handle the actual, tangible (manifest) mechanical and other like resources — as well as customers. In the case of the railways this staff is comprised from train staff: engineers, sleeper attendants, etc., station staff: train dispatchers, shunting staff, etc., passenger service staff: seat reservation and ticketing staff, etc. As do the managers, the ground staff must carry out actions according also to Rules & Regulations. And they may fail or succeed, more-or-less 'punctually & precisely'. Also this may be describable, informally, and perhaps also formally. Experiments and experience will show!

  To illustrate an issue we take up the thread from the bank example above.

  * **The Bank Example — Continued:**

    A clerk may perform the transaction correctly (and many different sequences of actions may be involved and applicable), or the clerk may make a mistake, or the clerk — or some already installed software support — maliciously diverts sums to "other" accounts! The contract therefore, in the domain, denotes a set of named *rule & regulation* designations. Each such named *rule & regulation*, since it may be potentially interpreted in any number of ways, is now modeled as a set of scripts. Transaction processing, say be a human clerk, then involves a non-deterministic choice among the possibly infinite ways of interpreting a client request.

```
type
  Acc = ...
          ...
        ∪ {scripts} m→ (Sn m→ Script-infset)
          ...

value
  int_Script: C × Trans → Bank → Bank
  int_Script(c,sn,vall)(b) ≡
    let a = (b(accounts))(c) in
    let ((pl,cl),cmd) = select((a(c))(sn)),
        env = (a(c))(env) in
    /* assert: */ len vall = len pl /* end assert */
    let ρ = [pl[i] ↦ vall[i] | i:Nat • i ∈ inds]
          ∪ [cl[i] ↦ env(cl[i])] in
    int_Cmd(cmd)ρ end end end

  select: Script-infset ~→ Script
  select(ss) as s post s ∈ ss
```

- User Sub–facets:

  Users (clients) interact with ground staff and with equipments (products) and service offerings of the domain system. They may interact according to expectations, or they may fail. They may be satisfied, or disgruntled. They may be loyal customers, or they may search for other 'vendors' of services and products in a competitive manner.

- *&c.*

Our list of facets have "moved" from the seemingly more easily formalisable, the "hard" facets, to "softer" facets that are increasingly more difficult to formalise.

- **Modeling:**

  The intrinsics, support technology and rules & regulations descriptions (models, whether informal or formal) must therefore be extended (enriched) to include the components, actions and behaviours of *humans*.

  To model, informally and formally, stake-holder facets may be difficult — but that is no reason for not trying. It seems that more research is needed, especially in the area of formalisation and in the concordance of informal and formal descriptions. That research may result in altogether different syntactical (visual) forms of descriptions.

There are many other (customer, economics, etc.) facets. It is not the purpose of this paper to enumerate as many as possible, nor to further analyse the concept of facets.

The list of facets given above is illustrative. The developers may be guided by this list, or may have to analyse the problem domain in order to determine for themselves the nature of other, not exemplified, facets.

2.6 Domain Elicitation & Validation

The terms elicitation and acquisition are used interchangeably.

There is an emerging, "rich" literature on techniques and tools that might help domain developers in extracting domain knowledge from stake-holders of the domain.

We would have liked, at this place, to give a reasonably thorough survey of contributions made by researchers and practitioners in the area. The problem is, however, that there are very few — if, in reality, any — relevant contributions. "Classical" software engineering tends to have focused on requirements elicitation and to have bundled occasional domain elicitation with requirements elicitation. An examination of the "schools" of domain knowledge engineering seems more relevant for our purposes.

Instead, therefore, of a satisfactory account we shall just mention a few possibly relevant papers:

- Formal Ontology: [52],
- Epistemology: [125]
- The knowledge–level reinterpreted: modeling socio–technical systems: [51]
- The frame problem in the situation calculus: [126]

  A simple solution (sometimes) and a completeness result for goal regression:
- Formal Ontology, Conceptual Analysis and Knowledge Representation, and: [79]

  Some Organising Principles for a Unified Top–level Ontology: [80]
- Modelling and Methodologies for Enterprise Integration: [13]
- The Logic of Enterprise Modelling: [78]
- An Algebraic Logic for Concept Structures [120]

  A Categorical View on Concept Structures [122]

  and Object Logic for Conceptual Modelling: [121]

  — with a Medical Domain as Case Study

We hope sometime to be able to relate this work to that of ours.

2.7 FAQ: Domains

- *Can stake]–holders understand the domain descriptions?*

  Appropriate stake-holders should understand corresponding perspectives of the informal descriptions. In fact it is desirable — in future, after computing scientists have identified basic methods — that they be able to write informal domain descriptions.

  Whether these stake-holders also can read the formal descriptions is another matter. We do not think that it is — at the moment — necessary that all classes of stake-holders meet this criterion.

For certain developments the client may make use of independent software engineering consultants (who can indeed both read and write formal descriptions) to inspect the developers documents — much like Norwegian Veritas and Lloyd's Register of Shipping act on behalf of future ship-owners when the ship is built.

- *What should be the languages of informal descriptions?*

  We believe they should be the languages spoken by the staff and users (customers) of the domain.

  In the example of railways this means that a variety of informal, yet sufficiently precise, professional languages should be used in a "cleaned-up" manner. The clean-up should only affect the non-professional, usualy, national and natural language parts and consists in improving the narrative and terminological precision.

  The informal, professional languages often deploy various diagrammatic parts (pictures, figures, tables) as well as sometimes even mathematical formulas. Such parts should be 'ported' to the narratives, etc.

- *What should be the languages of formal descriptions?*

  In this paper we show only the formal specification language of RSL [76], the RAISE [77] Specification Language. RSL is not the only possibility: we could probably as well have used VDM-SL, Z, or some other sufficiently endowed language. We do find, however, that RSL's concurrency constructs (not found in VDM-SL and Z) as well as its clear and simple methodology [77], bias us in the direction of RAISE.

  Where the domain exhibit temporal notions then RSL, VDM-SL and Z cannot be used — for those, temporal parts. Instead we might decide on using a suitable Duration Calculus.

  Many formal specification languages exists:

  - **B** [5, 3, 91, 4],
  - **Duration Calculi** [46, 50, 49, 43, 44, 92, 48, 93],
  - **Larch** [82, 83],
  - **RAISE/RSL** [76, 77]
  - **STeP/React** [114, 115],
  - **VDM** [24, 25, 104, 55, 109, 58, 108, 64],
  - **Z** [86, 139, 129, 135, 117, 132, 131, 136, 137, 138, 56, 40, 130, 133],
  - etc.

- *When have we specified enough — minimum/maximum?*

  Recall that domain description aims not primarily, but only also to serve as a basis for requirements description. That is: if we were only to describe the instantiated domain that very explicitly relates to requirements — we may call this kind of domain description 'minimal', then it is not so difficult to know when we have specified enough: We have specified a minimal domain when all the professional domain (system) terms that "pop-up" in requirements have been defined in the domain. But usually that "minimum" is insufficient for a number of reasons. 'Minimum' terms may need clarifications

which refer to undefined domain terms. Any one domain may give rise to several requirements, each covering (supporting) more-or-less "disjoint" areas of domain activity. Eventually emerging (i.e. resulting) software packages that implement these different requirements are desired to share facilities and concepts, i.e. to exchange data and 'call' each other! Any "gap" between the software packages usually is a reflection on some similar gap in their counterpart "minimal" domain descriptions. 'Domain-describing' these gaps — perhaps already before the software package interactions might have been conceived — amount to "non-minimal" domain descriptions. The process of securing a suitably comprehensive domain description is an uncertain one.

We take the position wrt. to the above "minimality/maximality" problem, that it is an issue of normative versus instantiated domain descriptions: minimal when instantiated, maximal when normative!

- *Normative and/or Instantiated Domain Descriptions?*

  - **Normative Domain Descriptions:**

    A normative domain description is a description which is intended to describe a class of usually two or preferably more "closely resembling domains". A normative railway domain description should thus cover for example the railways of Denmark, Norway, Sweden, perhaps even Russia and China — in fact: should desirably describe any national or private railway system! Whether such a normative description is possible is another matter!

    So a normative description may ideally cover the class of all domains, especially domain systems and their environments, but will probably do so in a way that makes their use for any particular, any specifically instatiated domain a less than trivial, but not an altogether unreasonable task.

    Research into and the development of such normative domain descriptions may typically not be a concern of any one particularly instantiable domain (system): why should they develop more than they think they need? Why, in competive situations, should they develop something that might as well benefit competition? Etcetera. So we may conclude that if it is reasonable to develop normative domain descriptions, then the needed precursor research as well as the development ought take place in peer-reviewed contexts, in an open fashion, that is: typically at a public research center or at a university.

    One can therefore imagine a potentially many year university project, with internationally collaboratoring "schools" — with varying participation over the years. To develop a reasonably comprehensive, normative model of a typical infrastructure domain may take 10–20 years. As in nuclear physics, the domain model emerges through partial contributions, slowly, but steadily.

    We suggest such a possibility for a number of domains: railways (residing in transport departments or institutes at technical universities), financial service industry (residing at schools of economics, finance and business management), etc. There is already loose collaboration between individuals of such schools, but perhaps 'human genome'-like 'domain projects' could be justified.

  - **Instantiated Domain Descriptions:**

> Since there can be little if any doubt that any specific domain (or system within such) needs domain descriptions that are particularly "geared" to its "peculiarities", there can also be little doubt that it would be nice if there were already available a normative, and appropriate domain description from which a rewrite, editing or parameterisation into an instantiated domain description was a reasonably straightforward development step.

We foresee a day when the description methods, techniques and tools of computing science and software engineering have matured to such a degree and relative to a number of domains such that continued methdology research and tool development takes place, not in computing science and software engineering departments, but in the more domain specific institutes. This predicated situation is akin to that of numerics, in fact of classical mathematics: many branches of natural sciences and engineering are today themselves capable of conducting necessary and sufficient mathematical work on modeling their own domains.

- *Why Domain Engineering by Computing Scientists & Software Engineers?*

  If we examine the basic development issue across the spectrum of domain, requirements and software design engineering (especially for man-made domain systems, in particular infrastructure systems), then we find that the overwhelmingly largest construction tasks all have to do with structuring very large descriptions: securing proper syntax, semantics and pragmatics. These descriptions shall primarily satisfy laws of mathematics, in particular of mathematical logic.

  No other engineering focuses so intensely on textual structures. No other engineering discipline speaks of syntax, semantics and pragmatics. In all other engineering branches there is sooner or later a quantum jump: from some diagrammatic, computable description to the (assembly line or refractory tower or other) construction of tangible, manifest products satisfying laws of nature.

  A major contribution of computing science and software engineering is exactly that of devising precise techniques and tools for handling large descriptions.

  That is the reason why computing science must study and software engineering must practice domain engineering — for years to come.

2.8 Domain Research, Education and Development Issues

The Bernried workshop, sponsored by the US Department of Defense (DoD) Office of Naval Research (ONR), had as a main objective of the ONR to evaluate, on the background of workshop presentations and discussions, which were and are the research, education and development issues. In this section, and in sections 3.5 (page 27) and 4.6 (page 31), we therefore relate put own contribution to that of needed research, education and development.

- **Domain Research Issues:**

  We need do more research on the linguistic and formal domain recognition and capture issues that may govern both informal and formal descriptions of domains [98].

  Specialisation in software engineering is one way of achieving the level of other engineering disciplines' professionalism and methodology. We may most likely be well-adviced

in moving firmly in that direction by following Jackson's notion of Problem Frames [98, 99, 36].

To do this research we envisage a number of desirably parallel executed experimental research projects. The number and kind of these should preferably be chosen so as to span a suitable spectrum of problem frames, and within the specifically chosen (problem frame specific) examples also span a suitable variety of perspectives and facets.

We need sharper, more methodology-oriented, perhaps formally founded and explored, ways of characterising the perspective and facet concepts, as well as individual such perspectives and facets. For a beginning we may follow the software view notion of Daniel Jackson [95].

- **Domain Education Issues:**

  There are currently no appropriate text books and monographs in the area of domain knowledge engineering, but there is are papers on knowledge engineering, ontology and enterprise modelling.

  The current author is issuing a series of reports covering the spectrum from domain engineering via requirements enginering to software design. These are intended to also be part of a monograph on formal aspects of software engineering.

- **Domain Development Issues:**

  Much experimental and exploratory development is needed in order to ensure that the researched and evolving domain modeling techniques and the concepts of perspectives and facets are appropriate. Over the years 1994–1997 we explored domain models for railways [26, 37, 47, 62, 35], manufacturing industry [74, 75, 6, 102, 7, 101, 103], ministry of finance [57], etc. while we established, built up and directed UNU/IIST, the UN University's International Institute for Software Technology. We are currently, 1997–1998, at our current address, with colleagues and students, further exploring domain methodologies in the areas of railways, metropolitan transports, banking, full scale finance accounting [28, 107, 21], etc.

3 Requirements

The aim of requirements enginering is to develop, together with stake-holders of the selected domain, a precise set of concordant descriptions of the requirements, a set that the parties, developers and stake-holders, can agree upon.

We will not give a detailed account, such as in the revious section on domain engineering, but only touch upon some issues.

Thus we expect a set of precise contractual obligations binding the two parties.

From a formal point of view, requirements engineering establishes the theory $\mathcal{R}$.

3.1 Requirements Models

Without further ado we — perhaps somewhat dogmatically — state that a requirements specification, i.e. a requirements model, basically builds upon the domain model, possibly a subset. *Requirements reside in the domain* as Zave & Jackson says [143].

3.2 Requirements Aspects

* **Requirements Aspect:**

  We can (formally) define an aspect as a partial specification of a requirements, consisting of a type space and a set of functions.

A number of aspects seems to govern the composition of requirements:

- **Domain Aspects — 'Functional' Requirements:**

  - **Domain Projections:**

    Some of the domain type (i.e. state) space is usually projected onto the requirements.

    From the domain model we have:

    type
    rTraffic = T $\xrightarrow{\sim}$ (Tid $\xrightarrow{m}$ Rou)
    oTraffic = T $\xrightarrow{m}$ (Tid $\xrightarrow{m}$ Uni)

    That is: "real traffic" is a partial function (total over a closed time interval) from time to the discrete routes of the railway net occupied by identified trains. Observable traffic discretises (through 'sampling') the partial function.

    This was in the domain. Now, if the requirements have to do with monitoring the air traffic, then we must decide upon (i) what, more precisely, of the air traffic is being observed, (ii) how often (i.e. more precisely about the 'sampling'), and (iii) by whom (i.e. how).

    From an underlying reality of support technology of sensors one could imagine that what we are observing is more like:

    type
    Interval = T × T
    Sensing = Interval $\xrightarrow{m}$ (Uni $\xrightarrow{m}$ Tid)

    hence

    value
    Convert: Sensing $\xrightarrow{\sim}$ oTraffic

    The requirements have to deal with domain component (here state) projection issues related to this.

    Similarly with the functions that may need be computed by the software:

    value
    TooClose: oTraffic $\xrightarrow{\sim}$ (Tid $\xrightarrow{m}$ Uni)
    Crashes: oTraffic $\xrightarrow{\sim}$ (Tid $\xrightarrow{m}$ Uni)

- **Domain 'Dichotomies':**

  Here we take a dichotomy as a potential conflict between what is expressed in one part of the domain model and what is expressed in another part of that domain model. Dichotomies only need be resolved by a requirements if the general requirements otherwise relate to the concerned parts of the domain model.

  An example is that of "ghost trains". If the requirements is about monitoring (and 'controlling') traffic, then the Law of N 'Ghost' Trains of the intrinsics somehow conflicts with the possibility of failing support technology to create the illusion, through misleading samplings, that there are indeed 'ghost' trains. On one hand we know that the intrinsics espresses that there can not be 'ghost' trains, while on the other hand we might indeed register such!

  A requirements to a traffic monitoring system may be to resolve such conflicts through re-sampling, and — if such fails — to correct the illusion, for example by stopping appropriate trains.

- **Domain 'Extensions':**

  Functional requirements usually focus on some domain concepts and facilities and direct the support of some of these. Once the whole apparatus of for example extensive and expensive, net-wide train sensing is being demanded, it may be little extra to demand that a number of traffic prediction and rescheduling functions also be required: functions that were not in the domain because they were impractical or inordinately expensive to realise without computing.

  In other words: the software + hardware machine, once inserted into the domain, becomes a part of it, and its concepts and facilities becomes a part of the domain for the next round of development.

- **Machine Aspects:**

  Among so-called "non-functional"[24] requirements we have those that relate to the machine itself, where by the machine we mean the computing systems made up by the required software and its execution platform, both soft and hard. Aspetcs include:

  - **Execution Platforms:**

    Requirements may dictate the use of specific hardware as well as run-time system software such as operating system, database management system, data (network) communication software etc. Among "etc." we include OMG packages (CORBA), etc.

  - **Dependability & Performance Issues:**

    For the specific combination: (i) provision of functional aspects (concepts and facilities), and (ii) computing platform, the client usually expects a certain quality of dependability & performance:

    * `Availability`: minimum down–time
    * `Reliability`: mean time between and to next failures, etc.,
    * `Safety`: machine response in case of equipment failures,

[24] The functional requirements are those (formalisable) ones that derive directly from (i.e. 'reside' in) the domain.

* `Security`: hierarchies of authorised access to the use of facilities, and:
* `Performance`: execution times needed in order to provide timely computations of certain functions; and: response times expected wrt. domain interactions.

Some of these aspects may be formalisable, others not (yet).

- **Maintenance Issues:**

  Perfective, corrective and adaptive maintenance is unavoidable. Requirements (actually they are a kind of meta-linguistc requirements) may make statements as to the "ease" with which such maintenance can be performed.

  Perfective maintenance aims at improving performance. Corrective maintenance aims at removing bugs. Adaptive maintenance aims at fitting existing software to new hardware and/or new software extensions to previously required software.

  As in our response to user-friendliness requirements below, we argue now that carefully developed and sufficiently broad domain models help us to "anticipate software sockets" for next generation software packages within the domain. And thus to help improve adaptability.

- **Domain ↔ Machine Interface Aspects:**

  Among further, so-called "non-functional" requirements we have those that relate specifically to the man/machine interface.

  - **Graphical & other User Interfaces**

    Our view is here: Visualising clear, well-described domain concepts determine basic concepts of graphical interfaces.,

  - **Dialogue Monitoring & Control**

    Again our view is: clear, well-described domain concepts, including events and behaviour, determine basic concepts of dialogue management.

  - **User Friendliness: Psychology, Physiology, etc. of Interface:**

    Often a broad sweeping statement is made: *"the software, when the basis of executions, should lead to a user-friendly system"*.

    As in our response to adaptability requirements above, we repeat the argument that carefully developed and sufficiently broad domain models where the eventually developed software is expected (required) to primarily reflect only the concepts and facilities of the domain, in some isomorphic or homomorphic manner[25], is an indispensable basis for securing user friendliness.

3.3 Requirements Elicitation & Validation

The terms elicitation and acquisition are used interchangeably.

I have little to say on this subject — and, although I have indeed followed the literature on requirements [134, 60, 63, 105, 53, 124, 116, 15, 127, 113, 111, 88, 54, 141, 94, 87, 106, 61]

[25] A step of development from one, abstract step to a concrete step can be said to be homomorphic if individual concepts of the abstract step are likewise individually identifiable in the concrete step.

I have much to do as I find that most of the literature need be re-conceptualised, re-worked and re-worded substantially to relate meaningfully in a formal and domain-based setting.

Fine insight need be rather substantially revised when requirements, as here suggested, are separated, but derived from domains.

Immediate, fine candidates, in my rather personal view, are: [73, 118, 14, 97, 110, 39, 143, 112].

3.4 FAQ: Requirements

- *Where do requirements come from?*

  From the domain, and basically only from the domain. The functional requirements are expressed by stake-holders in the domain and in terms of their professional language.

- *What about platform and interface requirements?*

  Well they are usually domain-independent. An example are software correctness. If computers etc. already exist in the domain, platform requirements and interface usually relate to these.

- *Should clients read formal requirements document?*

  The answer is along the same line as given above — FAQ: Domains — first item.

- *Requirements always change, so why formalise?*

  Well, requirements do change, but the domain from which they emerge change much less rapidly. Therefore it is additionally useful (i) to try "complete" a domain specification, (ii) to formulate requirements using terms only from the domain, and (iii) to base a software architecture on the core concepts of the domain, and hence the requirements. For situations where the application problem "occupies" but a tiny fraction of the domain it is usually still useful, in anticipation of future requirements changes, to 'relate', in the requirements and in the software architecture, "back" to the larger concepts of the domain. By mandating that the requirements 'homomorphically' reflect the domain, and that the software design 'homomorphically' reflects the requirements, some considerable robustness is achieved — and one can calmly await and handle requirements changes.

3.5 Requirements: Research, Education and Development Issues

For the issues listed below answers in line with those of section 2.8 (page 22) can be given here.

- **Requirements Research Issues:**

  We need better understand the relations between domains and requirements and between requirements and software architecture. The issues of projection, of the input, vetting and update of domain (data value) projections need also be studied. Finally the issue of relations between functional and non-functional requirements need skeptic clarification. In particular we need to better understand whether non-functional requirements can be formalised. To this end one may need to investigate entirely new (formal) specification paradigms.

- **Requirements Education Issues**

  Current textbooks in requirements engineering are full of many very good, mostly pragmatic observations, but I believe that we need textbooks on requirements engineering (etc.) that use formal specification and design calculation techniques [144, 128].

- **Requirements Development Issues:**

  We refer to the domain item on this issue page 23.

4 Software/Systems Design Engineering

An aim of design engineering is to develop, for the client, a software package or system (i.e. a set of "connected" packages) that satisfies the requirements.

Usually, given a requirements specification we can normally design both the hardware configuration and the software system — so we may take the term 'systems engineering' to include both hardware and software (sub–)system design.

A legally binding contract between the developer and the client describes mutual obligations wrt. delivery of ("more-or-less") correct software.

From a formal point of view, design engineering establishes the software design theory $\mathcal{S}$.

4.1 Software Architecture

By a software architecture we understand a specification which primarily specifies the external interface behaviour of the software (to be, or already designed).[26] In contrast to (the) 'program organisation', the software architecture, typically, implements the functional requirements whereas, typically, implements the non-functional requirements.

The important aspect of software architectures that we need to focus on here is that they are basically derived from the requirements.

Indeed, it can sometimes be a bit difficult to see any deeper difference between a requirements specification and a software architecture specification. We have found, however, that the following characterises the step from requirements to software architecture:

[26] David Garlan et al. ([8, 71, 1, 72, 9, 2, 69, 10, 70]) define the concept of Software Architecture much more broadly than we do. We do not mind, but find it a little disturbing. In the 1960s computer (ie. hardware) architecture was agreed, and was defined, seminally, by Amdahl, Blaauw and Brooks [11], to be the interface as seen by programmers: the computer data structures (byte, halfword, word, double word and variable length character strings), the addresing forms, the instruction repertoire, the channel commands, etc. That is: All things that were visible at the assembler code level. In 1964 the one IBM/360 architecture gave rise to a variety of machine organisations: from the IBM 360 Model 20, via Models 30, 40, 44, 50, 60, 65 and 70; that is: from byte via halfword, and word to double word machine data busses, from strictly sequential machines to highly overlapped (pipelined) flows. So: when it came to hardware there was a clear distinction between the architecture and the organsiation — just as we have seen with the Intel (etc.) series from eight bit byte organsiations (8086) via halfword, and word to double word machine data busses. But with basically the same architecture. We also refer to Hennesey and Pattersons two tomes: [90, 89]

Instead it seems that Garlan et al. defines an architecture to be anything you would like to say aobout the structure of either soft– or hardware and such that you say this diagrammatically.

We do not mind their definition, only — in relation to our software related definitions — theirs mean: any form of software architecture, program organisation or more refined structure.

- **Requirements as a Set of Partial Specifications:**

  — in contrast to a software architecture specification which collects all the "bits and pieces" of the various, aspect-oriented partial specifications.

  The software architecture specification therefore formulates a consistent and complete whole.

- **Requirements as a Under-specified Specification:**

  — in contrast to a software architecture specification which completes the requirements: fills in "wholes" that were deliberately left under-specified.

  It is often useful to let some requirements facet specifications be completed during software design.

- **Functional vs. Non-funcional Requirements:**

  Functional requirements usually can be rather explicitly "carried" into software architectures — as they were usually also formally specificed.

  Non-functional requirements are usually not (yet) formalisable. A software architecture — or, at the "latest", a program organisation — proposal therefore has to come up with initial answers as how to satisfy these non-functional requirements (such as performance, security, user-friendliness, maintainability, etc.).

4.2 Program Organisation (Software Structure)

By a program organisation (or [internal] software structure) we understand a specification which, in addition to the externally observable interface behaviour also specifies the internal structuring of the software (to be, or already designed). A determining factor in choosing one organistation design over another is whether non-functional requirements can thereby be satisfied.

A program organisation thus settles many issues that might have been left 'abstract' even by the software architecture. Examples are: A software architecture may specify a data type abstractly. A concretisation of this seemingly 'monolithic' abstract data type may be in the form of a set of data types. The program organisation specification further commits each member of the set to be implemented as a state variable (i.e. as an assignable variable) — and these may then be [geographically] distributed.

And a program organisation, in line with the above, introduces and specifies internal processes, committed (concrete) data structures — including the use of for example database management system support, data communication system support — etc.

4.3 Refinement

Although an element of software development we need not treat this methodology concept in this paper — since we primarily wishes to relate domains to requirements, requirements to software, and since we also primarily wish to enunciate the concepts of domain perspectives, domain facets, requirements aspects and software views.

4.4 Software Views

[95] defines a:

- **Software View:**

  as a partial specification of a program, consisting of a state space and a set of operations.

We have "re-used" this definition, slightly paraphrased, in our characterisations of domain perspectives and facets and requirements aspects.[27]

Since Daniel Jackson has basically set the agenda for the study of software views we shall refer to his paper [95].

4.5 FAQ: Formal Software Design

Instead of listing frequently asked questions wrt. software design we list a number of myths and commandments more generally related to the larger concept of 'formal methods':

- In [84] Anthony Hall lists and dispels the following seven "Myths":

  1. *Formal Methods can Guarantee that Software is Perfect*
  2. *Formal Methods are all about Program Proving*
  3. *Formal Methods are only Useful for Safety-Critical Systems*
  4. *Formal Methods Require highly trained Mathematicians*
  5. *Formal Methods Increase the Cost of Development*
  6. *Formal Methods are Unacceptable to Users*
  7. *Formal Methods are Not Used on Real, Large-Scale Software*

- In [41] Jonathan P. Bowen and Michael G. Hinchey continue dispelling myths:

  8. *Formal Methods Delay the Development Process*
  9. *Formal Methods are Not Supported by Tools*
  10. *Formal Methods mean Forsaking Traditional Engineering Design Methods*
  11. *Formal Methods only Apply to Software*
  12. *Formal Methods are Not Required*
  13. *Formal Methods are Not Supported*
  14. *Formal Methods People always use Formal Methods*

- And in [42] Jonathan P. Bowen and Michael G. Hinchey suggests ten rules of software engineering conduct:

  I. *Thou shalt choose an appropritae notation*

  II. *Thou shalt formalise but not over-formalise*

  III. *Thou shalt estimate costs*

  IV. *Thou shalt shall have a formal methods guru on call*

[27] Pages 11, 12 and 24 respectivey.

V. *Thou shalt not abandon thy traditional development methods*

VI. *Thou shalt document sufficiently*

VII. *Thou shalt not compromise thy quality standards*

VIII. *Thou shalt not be dogmatic*

IX. *Thou shalt test, test, and test again*

X. *Thou shalt reuse*

4.6 Software Design: Research, Education and Development Issues

The:

- **Software Design Research**
- **Software Design Education**
- **Software Design Development**

issues seem reasonably well taken care of in at least Europe. The European, so-called 'formal methods' awareness "movement" (as exemplified through the more than a decade-long efforts of first VDM Europe, later Formal Methods Europe (FME)) These propagation efforts are based primarily on European research.

The US attitude is basically that formal methods are anchored in, yes some (John Rushby) even state: only have to do with tools. The European attitude, in contrast, take formal methods are mostly specification (i.e. formal specification and design calculi).

It will be interesting to see how these two schools may eventually merge.

The US school on 'software architecture', notably that part which we call: 'program organisation', is very strong [8, 71, 1, 72, 9, 2, 69, 10, 70]. We should like to see a clearer separation between what we define as separate concepts: software architecture and program organisation. Some research is needed to clarify this issue and to develop principles and techniques for the 'derivation' of (families of) architectures from requirements specifications and of (families of) program organisations from these architectures.

5 Conclusion

A proposal for a triptych decompostion of software engineering has been presented. Some of the subsidiary, methodology principle concepts has been likewise presented: domain principles and facets, requirements aspects and software design views. A development methodology assumption is that all descriptions being presented (both informally and formally), and that relations between triptych phase documents and between stages and steps within these, be also formally characterised.

The paper has suggested a number of software engineering practices be currently dispensed by software engineers rather than domain professions. We argue so since the disciplines of computing science and software engineering has carefully developed and honed attendant description principles, techniques and tools. The paper has likewise suggested that a number of subsidiary areas be subject either to research, and/or to support by more or less mechanised tools, and/or to more specialised education: teaching and training.

What has been described is essentially the authors current research, university education and technology transfer interests. With colleagues we are trying out ideas of this paper in student project work, in exploratory & experimental demo & prototyping work — some with one or another of the kind of infrastructure enterprises or industries mentioned in section 2.1 — and hence also in more or less applied research. It is therefore to be expected that future publications will report on this as well as on more foundational work.

A Software Engineering Terminology

A.1 Special Terminology

The wording of many of the definitions of this report may sound dogmatic. Prudent reflection will soon reveal that it is merely a set of reasonable and useful delineations.

1. Software Development:

   To us software development consists of three major components: domain engineering, requirements engineering and software design. Together they form software engineering.

   > Discussion: *This is a somewhat "bureaucratic" characterisation. Namely one given in terms of its "way of being handled" — who does it, rather than what it does!*

   Therefore: Software Development aims at constructing software — or as we shall later "enlarge" it: machines. It does it by also constructing models of the domain in which the software will reside, the requirements that the software must satisfy, etc. The present report will deal with the processes of software development.

2. Systems vs. Software Engineering:

   Perhaps the term 'software engineering' is too restrictive. Since any implementation of especially a larger software system entails procurement also of hardware, development will also include configuration and acquisition of hardware components. That larger concept: the development, procurement, installation, performance tuning, operation and disposal of computing systems (hardware ⊕ software) is therefore what we mean by systems engineering. Thus software engineering is part of systems engineering.

   > Discussion: *As eloquently pointed out by Michael Jackson [98] the term software engineering is probably much too broad a term, or it should be understood as a class term. As such it covers a set of specialised software engineer(ing specialtie)s. Mechanical engineering stands for rather separate groups of for example automotive, heat/water/ventilation, hydrological, nautical, aero-nautical, and many other engineering specialities. Software engineering is still far from having identified suitably specialised such groups — except perhaps for compiler designers. We refer to item 12 (page 41) for hints at what such groups might be.*

3. Linguistic Notions:

(a) Descriptions & Documents:

All stages of software development results in descriptions and documents. The two terms are almost synonymous: description refer to the semantic content of the syntactic document. We describe and document domains, requirements, software architectures, program organisations, etc. We sometimes also, again synonymously, refer to these descriptions as Definitions (as f.ex. for a domain model or a requirements model), sometimes as Specifications (as f.ex. for a software architecture model), yes even as Designs (as f.ex. for a program organisation model).

software engineering management takes the syntactic, document view of development; whereas programming takes the semantic, description view.

(b) Concordant Documents:

A set of documents, spanning the spectrum of descriptions of domains, requirements, software architectures, program organisations, etc., form a set of *concordant* descriptions, and within each of these we may also need alternative, complementary descriptions — which form another set of *concordant* descriptions.

Two or more documents are said to be concordant wrt. each other if they all purport to present descriptions of basically the same thing — but each emphasising different, but related aspects.

We shall later introduce pragmatic notions of perspectives, facets, aspects and views. These represent equivalence classes of concordant documents.

(c) The Informal Languages of Indications, Options and Actions:

As pointed out by Jackson [99] the informal language of domain descriptions is indicative: "what there is", that of requirements descriptions is optative: "what there should be", and that of software design descriptions is imperative: "do this, do that — how to do it!".

(d) Descriptive and Prescriptive Theories:

We could also use the terms descriptive and prescriptive theories in lieu of indicative and optative descriptions.

(e) The Formal Languages of descriptions:

In contrast, the languages of formal descriptions are mathematical, and in mathematics we cannot distinguish between indicative, optative and imperative moods. Such distinctions are meta-linguistic, but necessary. Similarly with the various equivalence classes of concordant documents: perspectives, facets, aspects and views.

(f) Description Techniques:

We refer to Jackson [98, 99]: "Phenomenology — *recognising* and *capturing the significant elementary phenomena* of the subject of interest (domain, requirements, software) *and their relationships. Say as much as is necessary, with perfect clarity, but no more. ... Choose and express abstractions and generalisations* formally *in order* necessarily *to bring an informal reality under intellectual control.*"

Constituent techniques [99] are those of:

- `Designations:`
  That is: system identification. Establishing the informal relationship between real world phenomena and their description identifiers.

- `Definitions:`
  The definition of concepts based on real world phenomena.
- `Refutable Assertions:`
  The usually axiomatic expression of real world properties.

4. Machine:

   The aim of software development is to create software. That software is to function on some hardware. Together we call the executing software ⊕ hardware for the machine. The machine is, in future, to serve in the (future) domain as part of the (future) system.

   Since domain engineering and requirements engineering aim at descriptions that may eventually lead to procurement of both software and hardware we shall refer to software development leading to a machine.

5. Domain Concepts:

   Two approaches seem current in today's 'domain engineering': one which takes its departure point in model–oriented, Mathematical Semantics specification work (and which again basically represents the 'Algorithmic' school), and one which takes its departure point in knowledge engineering — an outgrowth from AI and Expert Systems. The latter speaks of Ontologies. For now we focus on the former approach.

   (a) Domain = System ⊕ Environment ⊕ Stake-holders:
   By domain we roughly understand an area of human or other activity. We "divide" the fomain into system, environment and stakeholder. All are part of a perceived world.

   `Discussion:` *Examples of domains are: railways, air traffic, road transport, or shipping of a region; a manufacturing industry with its consumers, suppliers, producers and traders; a ministry of finance's taxation, budget and treasury divisions as manifested through government, state, provincial and city offices and their functions; the financial service industry, or just one enterprise in such an industry (a bank, an insurance company, a securities broker, or a combination of these); etcetera.*
   Since we are developing software packages that serve in these domains it is important that the software developers are presented with, or themselves help develop precise descriptions (models, see later) of these domains.
   Our argument here parallels that given for compiler development: we must first know the syntax and semantics of the (source, target and implementation) languages involved.

   (b) System:
   By system we understand a part of the domain. The system is typically an enterprise. Once the machine has been installed in the system then it becomes a part of a new domain wrt. future software development.

   `Discussion:` *A railway System consists of the railway net (lines, stations, signalling, etc.), the rolling stock (locos, passenger waggons, freight cars, etc.) and trains, the time tables and train journey plans, etc. A description*

of the railway domain must make precise the structure and components of the railway systems as well as all the behaviours it may exhibit.
Identification of the system is an art.
Please note that when we speak of a system we do not refer to a computing system.

(c) Environment:

By environment we understand that part of the perceived world which interacts with the system. Thus the system complement wrt. "the perceived world", i.e. the environment, together with the system and stakeholder makes up the domain of interest.

Discussion: *The Environment of an air traffic system includes the weather (the meteorology) and the topology of the geographical areas flown over.*
Identification of the Environment is an art.
Since the Environment interacts with the System (and hence potentially with the Machine to be built) it is indispensable that we describe (incl. formally model) that part of the Environment which interacts.

(d) Stakeholder = Clients ⊕ Customers ⊕ Staff:

By stakeholder we mean any of the many kinds of people that have some form of "interest" in the (delivered) machine: enterprise owners, managers, operators and customers of the enterprise: within the system or in the environment.

Discussion: *Stakeholders of a ministry of finance include government ministers, ministry staff and tax payers,*
Identifying all relevant stakeholders is an art.
Since also they interact with the System (and hence potentially with the Machine to be built) it is indispensable that we describe (incl. formally model) possible stakeholder interactions with the System.

(e) Client:

By client we understand the legal entity which procures the machine to be developed. The client is one of the stakeholders, and must be considered a main representative of the system.

Discussion: *A financial enterprise Client is usually the appropriate level executive who specifically contracts some software to serve in the enterprise.*

(f) Staff:

By staff we understand people who are employed in, or by, the system: who works for it, manages, operates and services the system. staff are a major category of stakeholders.

(g) Customer:

By customer we understand the legal entities (people, companies), within the system, who enter into economic contracts with the the client: buys products and/or services from the client, etc. customers form another main category of stakeholders: outside the system, but within the domain.

Discussion: *We have identified important components of a domain. The software engineers — in collaboration with domain stakeholders — face*

the further tasks of specifically identifying the exact components to be considered for a given Domain.
That 'identification' is still an art: requires experience and cannot be settled before preliminary modelling experiments have been concluded.

(h) Domain Engineering
= Recognition
↔ Capture
↔ Model
↔ Analysis
↔ Theory:

Domain Engineering, through the processes of domain acquisition and domain modelling, establishes models of the fomain. A domain model is — in principle — void of any reference to the machine, and strives to describe (i.e. explain) the fomain as it is. domain analysis investigates the domain model with a view towards establishing a domain theory. The aim of a domain theory is to express laws of the fomain.

Discussion: *The Domain Engineer could be a special version of a Software Engineer — one who could be specially trained both as a Software Engineer, in general, and as a "Domain Expert", in particular.*

(i) Domain Recognition:
System identification is an art! To recognise which are the important phenomena in the domain, and which phenomena are not (important) is not a mechanistic "thing".

(j) Domain Capture
= Acquisition
↔ Modelling:

Discussion: *We make a distinction between the "soft" processes of domain acquisition: linguistic and other interaction with stakeholders, and domain modelling: the "hard" processes of writing down, in both informal and formal notations, the domain model.*
The domain capture process, when actually carried out, often becomes confused with the subsequent requirements capture process. It is often difficult for some stakeholders and for some developers, to make the distinction. It is an aim of this report to advocate that there is a crucial distinction and that much can be gained from keeping the two activities separate. They need not be kept apart in time. They may indeed be pursued concurrently, but their concerns, techniques and documentation need be kept strictly separate.

(k) Ontology:
What we call domain models some researchers call ontology — almost!
In the 'Enterprise Integration and in the 'Information Systems communities ontology means: "formal description of entities and their properties". Ontological analysis is applied to modelling the domain of (manufacturing) enterprises and such systems (typically management systems) whose implementation is typically database oriented.

(l) Domain Model:

By a Domain Model we understand an abstraction of the Domain.

Discussion: *Usually we expect a Domain Model, i.e. a Description of the Domain to be presented both informally and formally.*
The informal Description typically consists of a Synopsis which summarises the Model, a Terminology which for every professional term of the Domain defines that term, and a Narrative which — in a readable style — describes how the terms otherwise relate. The formal Model is then expressed in some formal specification language and can be subject to Calculations using a Design Calculi of that notation. The model thus presents the syntax, semantics and, possibly also, the pragmatics of terms of the Domain. Not the syntax and semantics of the professional language spoken by Staff of the Domain System, but just the crucial terms.

(m) Domain Modelling Techniques

Domain modelling usually proceeds by constructing a partial specification (type space, functions and axioms) for each of a number of domain perspectives and similarly one for each domain facet.

(n) Description Technology:

Crucial concepts in domain modelling include:

- system identification,
- i.e. enumeration of designations [99],
- formulation of definitions and
- expression of possibly refutable assertions.

The latter typically in the form of constraints on types and functions.

(o) Domain Perspective:

Domain perspectives reflect the conception of the domain business as seen by various stake-holders.

(p) Domain Facet:

Domain facets reflect some more 'technical, pragmatic decomposition' of the domain together with a 'separation of concerns'. Specification typically proceeds from intrinsic facets, via support technology facets and rules & regulations facets to staff facets, etc.

(q) Domain Model Analysis:

By Domain Analysis we understand informal and formal analyses of the Domain and of the resulting Model — whether informal or formal.

Discussion: *The purposes of the analyses can be to ascertain whether a component and/or its behaviour qualifies as a component (etc.) of the Domain, and for such included components analyses may reveal Model properties not immediately recognised as properties of the Domain. Note the distinction being made here: the Domain as it exists "out there", and the Model as an abstraction thereof and which "exists" on the (electronic) "paper" upon which the Model is represented.*

(r) Domain Theory:

The purpose of Domain Analysis is to also establish a Theory of the Domain, or rather: of the Models purported to represent the Domain!

Discussion: *Examples of theorems in a theory of railways could be: (1) (Kirschhoffs law for trains:) "Over a suitably chosen time interval (say 24 hours) the number of trains arriving at any station, minus the number of trains taken out of service at that station, plus the number of trains put into service at that station, equals the number of trains leaving that station"; (2) (God doesn't play dice:) "Two trains moving down a line cannot suddenly change place"; (3) (No Ghost Trains) "If at two times 'close to each other' (say seconds apart) a train has been observed on the railway net, then that train is on the railway net somewhere between the two original observation positions at any time between the two original observation times". Etc.*

Failure to record essential theorems may result in disastrously erroneous software.

Ability to identify and establish appropriate theorems is an art and takes years!

(s) Domain Model Validation

An informal process whereby informal and formal specification parts are related and where these again are related to the "real world" domain (system identification)

6. Requirements Concepts:

Requirements, as we have seen, form a bridge between the larger Domain and the "narrower" software which is to serve in the Domain.

(a) Requirements = System ⊕ Interface ⊕ Machine:

Requirements issues are either such which concern (i) machine support of the system, (ii) human (and other) interfaces between the system and the machine, or (iii) the machine itself.

Requirements describes the system as the stakeholders would like to see it.

(b) Functional & Non-Functional Requirements:

Functional requirements include the concepts and facilities to be offered by the desired software.Non-functional requirements emphasise such less tangible issues as performance, user dialogue interface, dependability, etc.

(c) Requirements Engineering
= Capture
↔ Model
↔ Analysis
↔ Theory:

Requirements Engineering, through the process of requirements capture, establishes models of the requirements. The "conversion" from requirements information obtained through requirements elicitation, via requirements modelling to requirements models is called requirements capture. Requirements Models are formally

derived from and extends domain models. Requirements Engineering also analyses requirements models, in order to derive further properties of the requirements.

Discussion: *We hope the reader observes the "similarity" in the components of domain engineering vs. those of requirements engineering.*

(d) Requirements Capture
= Elicitation
↔ Modelling:

Remarks similar to those under Domain Capture — item 5j (page 36) apply.

(e) Requirements Model:

A specification of the requirements. Usually in the form of a set of partial specifications, one for each requirements aspect.

(f) Requirements Modelling Techniques:

Requirements "reside in the domain", and are hence primarily projections of their type space and functions. Functional techniques deal with projections, resolving domain/requirements dichotomies and extending domains. Non-functional techniques deal with machine notions: computing platform, system dependability and maintainability, and with computer human interface issues: user-friendliness, graphic user interfaces, dialogue management, etc.

(g) Requirements Model Analysis:

By Requirements Analysis we understand informal and formal analyses of the Requirements and of the resulting Model — whether informal or formal.

Discussion: *The purposes of the analyses can be to ascertain whether a component and/or its behaviour qualifies as a component (etc.) of the Requirements, and for such included components analyses may reveal Model properties not immediately recognised as properties of the Requirements. Note the distinction being made here: the Requirements as it exists "out there" — among Stake-holders, and the Model as an abstraction thereof and which "exists" on the (electronic) "paper" upon which the Model is represented.*

(h) Requirements Theory:

The purpose of Requirements Analysis is to also establish a Theory of the Domain, or rather: of the Models purported to represent the Domain!

7. Software Concepts:

(a) Software Design
= Software Architecture Specification
↔ Program Organisation Specification
↔ Refinements
↔ Coding:

Software Design, through the process of design ingenuity, proceeds from establishing a software architecture, to deriving a program organisation, and from that, in further steps of design reification, also called design refinement, constructing the "executable code".

(b) Software Architecture:

A software architecture description specifies the concepts and facilities offered the user of the software — i.e. the external interfaces.

Usually functional requirements "translate" into software architecture properties.

(c) Program Organisation:

A program organisation description specifies internal interfaces between program modules (processes, platform components, etc.).

Usually non-functional requirements "translate" into program organisation design decisions.

(d) Refinement:

Design Refinement covers the derivation from the requirements model of the software architecture, of the program organisation from the software architecture, and of further steps of concretisations into program code.

8. Creation — Acquisition, Elicitation and Invention:

All stages and steps of the software development process involves creation: domain acquisition & domain modelling, requirements elicitation & requirements modelling, and design ingenuity. This human process of invention leads to the construction of informal as well as formal descriptions.

9. Systematic, Rigorous and Formal Development:

The software development may be characterised as proceeding in either a systematic, a rigorous or even, in parts, a formal manner — all depending on the extent to which the underlying formal notation is exploited in reasoning about properties of the evolving descriptions.

(a) *Formal Notation:*

By a formal notation we understand a language with a precise syntax, a precise semantics (meaning), and a proof system. By "a precise ..." we usually mean "a mathematical ...".

(b) *Systematic Use of Formal Notation:*

By a systematic use of formal notation we understand a use of the notation in which we follow the precise syntax and the precise semantics.

(c) *Rigorous Use of Formal Notation:*

By a rigorous use of formal notation we understand a systematic use in which we additionally exploit some of the 'formality' by expressing theorems of properties of what has been written down in the notation.

(d) *Formal Use of Formal Notation:*

By a formal use of formal notation we understand a rigorous use in which we fully exploit the 'formality' by actually proving properties.

(e) *Formal Method* $\approx$ *Formal Specification* $\otimes$ *Calculation:*

We refer to item 3 (page 43) for a definition of 'method'.

The methods claimed today to be formal methods may be formal, but are not methods in the sense we define that term! Since we do not believe that a method

for developing software: from domains via requirements, can be formal, but only that use of the notations deployed may be, we (now) prefer the terms: formal specification and calculation.

(f) *Design Calculi — or Formal Systems:*

By a design calculus we understand a formal system consisting of a formal notation and a set of precise rules for converting expressions of the formal notation into other such, semantically 'equivalent' expressions.

10. Satisfaction = Validation ⊕ Verification:

The domain acquisition and requirements elicitation processes alternate with domain modelling and requirements modelling, respectively, and these again with securing satisfaction.

(a) Validation:

In this report we are not interested in the crucial process of interactions between software developers (i.e. software engineers, which we see as domain engineers, requirements engineers and software designers) and the stakeholders. validation is thus the act of securing, through discussion, etc., with the stakeholders that the domain model correctly reflects their understanding of the domain.

(b) Verification:

Let $\mathcal{D}$, $\mathcal{R}$ and $\mathcal{S}$ stand for the theories of the fomain, requirements and software. Then verification:

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

shall mean that we can verify that the designed software satisfies the requirements in the presence of knowledge (i.e. a theory) about the fomain.

11. Software Engineering:

Software Engineering is the combination of domain engineering, requirements engineering and software design, and is seen as the process of going between science and technology. That is, of developing descriptions on the basis of scientific results using mathematics — as in other engineering branches — and of understanding (the constructed fomain of) existing (software) technologies by subjecting them to rigorous domain analysis.

12. Frame Specialisation:

Discussion: *In item 2 (page 32) we discussed the problem of software engineering being seemingly as a too wide field. And we hinted that specialisation might be a natural way of achieving a level of professionalism achieved in traditional engineering fields. In this item we will briefly introduce the concept of problem frames and give example of distinct such frames.*

A problem frame is well-delineated part of all the problems to which computing might be applied — such that this frame offers a precise set of principles, techniques and tools for software development, and such that this 'method' fits the frame "hand in glove".

- *Principal Parts and Solution Task*

Following Jackson [98, 99] we think of a (problem) frame as consisting of its principal parts and a solution task. The principal parts are (1.) the domain — which exists a-priori — and (2.) the requirements. The solution task is that of developing the software — something relatively new! Tackling an application problem consists initially of analysing it into a frame, including a multi-frame with clearly identified part-frames.

We explain a few frames and otherwise refer to [98, 99, 27]:

(a) *Translation Frame:*

The principal parts are: (i) two formalised languages (syntax and semantics), source and target; (ii) the concrete form of the syntactic representations of either: the source usually in the form of a BNF grammar for textual input, the target usually in the form of an internal ("electronic") data structure; (iii) user requests for compilation from source to target; (iv) the compiler; and (v) the translation function. (i-ii) form the domain, (iii-iv-v) the requirements.

The solution task now involves developing the compiler using a well-defined set of techniques and tools: lexical scanner generators, possibly error-correcting parser generators, attribute grammar interpreters, etc.

(b) *Reactive Systems Frame:*

The principal parts are (i) the dynamic (temporal, real-time) "real world"; (ii) its observable variables [output], (iii) its controllable variables [input]; (iv) user (or other system) requests for the monitoring and/or control of the "real world"; (v) the monitoring & control (software etc.) system; and (vi) the specific monitoring & control functions (optimisation, safety, dependability, etc.). Items (i-ii-iii) form the domain, (iv–vi) the requirements.

The solution task now involves control theoretic and real-time, safety critical software design principles, techniques and tools.

It seems that Jackson refers to the reactive systems frame as the environment-effect frame [98].

(c) *Information Systems Frame:*

The principal parts are almost as for reactive systems (i-ii) except that there is no desire for control, and the issues of safety criticality, real-time and dependability are replaced by (vi) (observable) information security and the need for usually "massive" information storage (for statistical and other purposes); (iv) the requests are concerned with the visualisation of observed information and computations over these; (v) the system is thus more of an information (monitoring) system; and (vi) the functions include specifics about the visualisation and other processing.

The solution task can perhaps best be characterised in terms of the principles, techniques and tools for example offered by Jackson's JSD method [96, 98].

(d) *Connection Frame:* See [98, 99, 27] for details.

(e) *Workpiece Frame:* See [98, 99, 27] for details.

(f) *Transaction Frame:* See [98, 99, 27] for details.

(g) *Multi-frame:* See [98, 99, 27] for details.

Usually a problem is not reducible to a single of the frames mentioned above (and some of these, due to requirements, often "overlap"). In such cases we have a multi-frame, a frame being best characterised in terms of hopefully reasonable well-delineated (sub-) frames.

(h) *&c.*

A.2 General Terminology

Many more terms are used in the subject field of this report: in its science and in its engineering. Sometimes with unclear meanings, and not always with the same meaning from paper to paper. We shall therefore try delineate also important general concepts.

Some dogmas:

1. Computer Science:

   Computer Science, to us, is the study and knowledge of the foundations of the artifacts that might exist inside computers: the kinds of information, functions and processes (i.e. type theory), models of computability and concurrency; bases for denotational, algebraic and operational semantics; specification and programming language proof theories; automata theory; theory of formal languages; complexity theory; etc.

2. Computing Science:

   Computing Science, to us, is the study and knowledge of how to construct the artifacts that are to exist inside computers. Successful computing science results in a useful programming methodology.

   The present report "falls", subject-wise, somewhere between computing science and software engineering.

3. Method:

   By a method we understand a set of *principles* of *analysis*, and for *selecting* and *applying techniques* and *tools* in order *efficiently* to *construct efficient artifacts* — here software.

4. Methodology:

   By methodology we understand the study and knowledge about methods. Since we can assume that no one software development method will suffice for any entire construction process we need be concerned with methodology.

5. Software:

   By software we understand all the documentation that is necessary to *install, operate, run, maintain* and *understand* the executable code, as well as that *code* itself and the *tools* that are needed in any of the above (i.e. including the original development tools).

6. Software Technology:

   By software technology we understand sets of software tied to sets of specific platforms. (By a platform we mean "another" machine!)

7. Programming:

   Programming is a subset of activities within software engineering which focus on the systematic, via rigorous to formal creation of descriptions using various design calculi.

8. Engineering:

   Engineering is the act of constructing technology based on scientifically established results and of understanding existing technologies scientifically.

9. Engineer:

   Engineers perform engineering and use, as a tool, mathematics. It is used in order to model, analyse, predict, construct, etc. software engineers reason about the artifacts they construct, be they (fomain, requirements, software architecture, program organisation, etc.) model descriptions (i.e. definitions or specifications) or program code.

10. Technician:

    Technicians use technologies: they compose, use and "destroy" them — without necessarily using mathematics.

11. Technologist:

    Technologists are technicians who manage technologies: perceive, demand, produce, procure, market and deploy technologies.

This report views software engineering as hinted above: As the act of going between science and technology, using mathematics — wherever useful.

B Bibliographical Notes

References

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

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

[3] J.-R. Abrial, M.K.O. Lee, D.S. D.S. Neilson, P.N. Scharbach, and I.H. Sorensen. The B-method (software development). In S. Prehn and W.J.Toetenel, editors, *VDM '91. Formal Software Development Methods. 4th International Symposium of VDM Europe Proceedings. Vol.2: Tutorials; Springer-Verlag, LNCS.* VDM Europe, 1991. .

[4] Jean-Raymond Abrial. *The B Book.* Cambridge University Press, The Edinburgh Building, Cambridge CB2 1RU, England, 1996.

[5] J.R. Abrial. The B Tool (Abstract). In *[38]*, pages 86–87, September 1988.

[6] Cleta Milagros Acebedo. An Informal Domain Analysis for Manufacturing Enterprises. Research Report 62, UNU/IIST, P.O.Box 3058, Macau, March 1996.

[7] Cleta Milagros Acebedo and Erwin Paguio. Manufacturing Enterprise Simulation: A Business Game. Research Report 64, UNU/IIST, P.O.Box 3058, Macau, March 1996.

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

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

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

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

[12] Dao Nam Anh and Richard Moore. Formal Modelling of Large Domains — with an Application to Airline Business. Technical Report 74, UNU/IIST, P.O.Box 3058, Macau, June 1996. Revised: September 1996. .

[13] Peter Bernus and Laszlo Nemes, editors. *Modelling and Methodologies for Enterprise Integration*, International Federation for Information Processing, London, UK, 1996 1995. IFIP TC5, Chapman & Hall. Working Conference, Queensland, Australia, November 1995.

[14] D. M. Berry. The importance of ignorance in requirements engineering. *Journal of Systems and Software*, 28(2):179–184, February 1995. .

[15] B. Biebow and S. Szulman. Acquisition and validation of software requirements. *Knowledge Acquisition*, 6(4):343–367, December 1994. .

[16] D. Bjørner. Prospects for a Viable Software Industry — Enterprise Models, Design Calculi, and Reusable Modules. Technical Report 12, UNU/IIST, P.O.Box 3058, Macau, 7 November 1993. ppendix — on a railway domain model — by Søren Prehn and Dong Yulin,
Published in *Proceedings from first ACM Japan Chapter Conference*, March 7–9, 1994: World Scientific Publ., Singapore, 1994.

[17] D. Bjørner. Federated GIS+DIS–based Decision Support Systems for Sustainable Development — a Conceptual Architecture. Research Report 61, UNU/IIST, P.O.Box 3058, Macau, March 1996. Draft.

[18] D. Bjørner. Models of Enterprise Management: Strategy, Tactics & Operations — Case Study Applied to Airlines and Manufacturing. Technical Report 60, UNU/IIST, P.O.Box 3058, Macau, January – April 1996. .

[19] D. Bjørner. A Software Engineering Paradigm: From Domains via Requirements to Software. Research report, Dept. of Information Technology, Technical University of Denmark, Bldg.345/167–169, DK–2800 Lyngby, Denmark, July 1997. .

[20] D. Bjørner. Models of Financial Services & Industries. Research Report 96, UNU/IIST, P.O.Box 3058, Macau, January 1997. ncomplete Draft Report.

[21] D. Bjørner. Towards a Domain Theory of The Financial Sevice Industry. Research report, Dept. of Information Technology, Technical University of Denmark, Bldg.345/167–169, DK–2800 Lyngby, Denmark, July 1997. .

[22] D. Bjørner, C.W. George, B.Stig Hansen, H. Laustrup, and S. Prehn. A Railway System, Coordination'97, Case Study Workshop Example. Research Report 93, UNU/IIST, P.O.Box 3058, Macau, January 1997. .

[23] D. Bjørner, C.W. George, and S. Prehn. *Scheduling and rescheduling of trains*, page 24 pages. Prentice Hall (?), 1997.

[24] D. Bjørner and C.B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[25] D. Bjørner and C.B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.

[26] Dines Bjørner. Prospects for a Viable Software Industry — Enterprise Models, Design Calculi, and Reusable Modules. Technical Report 12, UNU/IIST, P.O.Box 3058, Macau, 7 November 1993. ppendix — on a railway domain model — by Søren Prehn and Dong Yulin,
Published in *Proceedings from first ACM Japan Chapter Conference*, March 7–9, 1994: World Scientific Publ., Singapore, 1994.

[27] Dines Bjørner. Michael Jackson's Problem Frames: Domains, Requirements and Design. In Li ShaoYang and Michael Hinchley, editors, *ICFEM'97: International Conference on Formal Engineering Methods*, Los Alamitos, November 12–14 1997. IEEE Computer Society.

[28] Dines Bjørner. Models of Financial Services & Industries. Research Report 96, UNU/IIST, P.O.Box 3058, Macau, January 1997. ncomplete Draft Report.

[29] Dines Bjørner. A Software Engineering Paradigm: Domain + Requirements + Design Enginering. Department of Information Technology, Software Systems Section, Technical University of Denmark, DK–2800 Lyngby, Denmark. This monograph is currently under development. It contains extended versions of [36, 31, 32, 31, 30, 33, 34], 1997–1998.

[30] Dines Bjørner. Domain Engineering: a precursor for Requirements Engineering and Software Design. Research, Department of Information Technology, Software Systems Section, Technical University of Denmark, DK–2800 Lyngby, Denmark, 1997–1998. This report is one in a series [36, 31, 32, 33, 34] that covers the Software Engineering spectrum from domain via requirements to design engineering from the point of view of methodology, formal specifications and design calculi. The reports are precursors of main chapters in the currently evolving monograph [29].

[31] 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, 1997–1998. This report is one in a series [36, 32, 30, 33, 34] that covers the Software Engineering spectrum from domain via requirements to design engineering from the point of view of methodology, formal specifications and design calculi. The reports are precursors of main chapters in the currently evolving monograph [29].

[32] Dines Bjørner. Requirements as an Arbiter between Domains 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 one in a series [36, 31, 30, 33, 34] that covers the Software Engineering spectrum from domain via requirements to design engineering from the point of view of methodology, formal specifications and design calculi. The reports are precursors of main chapters in the currently evolving monograph [29]. This report is being written for possible inclusion in a possible 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.

[33] Dines Bjørner. Requirements Engineering: Modelling Techniques. Research, Department of Information Technology, Software Systems Section, Technical University of Denmark, DK–2800 Lyngby, Denmark, 1997–1998. This report is one in a series [36, 31, 32, 30, 34] that covers the Software Engineering spectrum from domain via requirements to design engineering from the point of view of methodology, formal specifications and design calculi. The reports are precursors of main chapters in the currently evolving monograph [29].

[34] Dines Bjørner. Software Architectures and Program Organisations: Their Development from Domain and Requirements Models. Research, Department of Information Technology, Software Systems Section, Technical University of Denmark, DK–2800 Lyngby, Denmark, 1997–1998. This report is one in a series [36, 31, 32, 30, 33] that covers the Software Engineering spectrum from domain via requirements to design engineering from the point of view of methodology, formal specifications and design calculi. The reports are precursors of main chapters in the currently evolving monograph [29]. The reports are precursors of main chapters in the currently evolving monograph [29].

[35] Dines Bjørner, Chris W. George, Bo Stig Hansen, Hans Laustrup, and Søren Prehn. A Railway System, Coordination'97, Case Study Workshop Example. Research Report 93, UNU/IIST, P.O.Box 3058, Macau, January 1997. .

[36] 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. This paper is one in a series [31, 32, 30, 33, 34] that covers the Software Engineering spectrum from domain via requirements to design engineering from the point of view of methodology, formal specifications and design

calculi. The reports are precursors of main chapters in the currently evolving monograph [29].

[37] Dines Bjørner, Dong Yu Lin, and Søren Prehn. Domain Analyses: A Case Study of Station Management. Research Report 23, UNU/IIST, P.O.Box 3058, Macau, 9 November 1994. resented at the *1994 Kunming International CASE Symposium*: KICS'94, Yunnan Province, P.R.of China, 16–20 November 1994. .

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

[39] B. Boehm and H. In. Identifying quality-requirement conflicts. In *Second International Conference on Requirements Engineering (Cat. No.96TB100037); Colorado Springs, CO, USA*, page 218, Los Alamitos, CA, USA, 1996. IEEE Comput. Soc. Press. .

[40] Jonathan P. Bowen. Formal Specification of the ProCoS/safemos Instruction Set. *Microprocessors and Microsystems*, 14(10):631–643, December 1990.

[41] J.P. Bowen and M. Hinchey. Seven More Myths of Formal Methods. Technical Report PRG–TR–7–94, Oxford Univ., Programming Research Group, Wolfson Bldg., Parks Road, Oxford OX1 3QD, UK, June 1994. Shorter version published in LNCS Springer Verlag FME'94 Symposium Proceedings.

[42] J.P. Bowen and M. Hinchey. Ten Commandments of Formal Methods. Technical report, Oxford Univ., Programming Research Group, Wolfson Bldg., Parks Road, Oxford OX1 3QD, UK, 1995.

[43] Zhou Chaochen. Duration Calculi: An Overview. Research Report 10, UNU/IIST, P.O.Box 3058, Macau, June 1993. Published in: *Formal Methods in Programming and Their Applications*, Conference Proceedings, June 28 – July 2, 1993, Novosibirsk, Russia; (Eds.: D. Bjørner, M. Broy and I. Pottosin) LNCS 736, Springer-Verlag, 1993, pp 36–59.

[44] Zhou Chaochen and Michael R. Hansen. Lecture Notes on Logical Foundations for the Duration Calculus. Lecture Notes, 13, UNU/IIST, P.O.Box 3058, Macau, August 1993.

[45] Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A Calculus of Durations. *Information Proc. Letters*, 40(5), 1992.

[46] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A Calculus of Durations. *Information Processing Letters*, 40(5):269–276, 1991.

[47] Zhou Chaochen and Yu Huiqun. A duration Model for Railway scheduling. Technical Report 24b, UNU/IIST, P.O.Box 3058, Macau, May 1994.

[48] Zhou Chaochen, Dang Van Hung, and Li Xiaoshan. A Duration Calculus with Infinite Intervals. Research Report 40, UNU/IIST, P.O.Box 3058, Macau, February 1995. ublished in: *Fundamentals of Computation Theory*, Horst Reichel (ed.), pp 16-41, LNCS 965, Springer-Verlag, 1995.

[49] Zhou Chaochen, Anders P. Ravn, and Michael R. Hansen. An Extended Duration Calculus for Real-time Systems. Research Report 9, UNU/IIST, P.O.Box 3058, Macau, January 1993. Published in: *Hybrid Systems*, LNCS 736, 1993.

[50] Zhou Chaochen and Li Xiaoshan. A Mean Value Duration Calculus. Research Report 5, UNU/IIST, P.O.Box 3058, Macau, March 1993. Published as Chapter 25 in *A Classical Mind*, Festschrift for C.A.R. Hoare, Prentice-Hall International, 1994, pp 432–451.

[51] W.J. Clancey. The knowledge–level reinterpreted: modeling socio–technical systems. *International Journal of Intelligent Systems*, 8:33–49, 1993.

[52] N.B. Cocchiarella. Formal Ontology. In H. Burkhardt and B. Smith, editors, *Handbook in Metaphysics and Ontology*, pages 640–647. Philosophia Verlag, Munich, Germany, 1991.

[53] A. Cucchiarelli, M. Panti, and S. Valenti. Supporting user-analyst interaction in functional requirements elicitation. In *First Asia-Pacific Software Engineering Conference; Tokyo, Japan*, pages 114–23, Los Alamitos, CA, USA, 1994. IEEE Comput. Soc. Press.

[54] P. Darke and G. Shanks. Stakeholder viewpoints in requirements definition: a framework for understanding viewpoint development approaches. *Requirements Engineering*, 1(2):88–105, 1996. .

[55] John Dawes. *The VDM-SL Reference Guide*. Pitman Publishing, 1991.

[56] Antoni Diller. *Z: An Introduction to Formal Methods*. Wiley, Chichester, UK, June 1990.

[57] Do Tien Dung, Le Linh Chi, Nguyen Le Thu, Phung Phuong Nam, Tran Mai Lien, and Chris George. Developing a Financial Information System. Technical Report 81, UNU/IIST, P.O.Box 3058, Macau, September 1996.

[58] Rene Elmstrøm, Peter Gorm Larsen, and Poul Bøgh Lassen. The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. *ACM SIGPAN Notoces*, 29(9):77–80, September 1994.

[59] Myatav Erdenechimeg, Richard Moore, and Yumbayar Namsrai. MultiScript I: The Basic Model of Multi-lingual Documents. Technical Report 105, UNU/IIST, P.O.Box 3058, Macau, June 1997.

[60] J.A. Hess et al. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU-SEI-90-TR-21, Carnegie Mellon University. SEI Software Engineering Institute, DoD, Pittsburgh, Penn., USA, 1990.

[61] R. Thayer et al., editor. *Software Requirements Engineering*. IEEE Computer Society Press, Los Alamitos, California, USA.

[62] Yu Xinyiao et al. Stability of Railway Systems. Technical Report 28, UNU/IIST, P.O.Box 3058, Macau, May 1994.

[63] A. Finkelstein. Tracing back from requirements. In *IEE Colloquium on 'Tools and Techniques for Maintaining Traceability During Design' (Digest No.180)*, London, UK, 1991. IEE. .

[64] John Fitzgerald and Peter Gorm Larsen. *Developing Software using VDM-SL*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 1RU, England, 1997.

[65] Dov M. Gabbay, C.J. Hogger, J.A. Robinson, and D. Nute, editors. *Deduction Methodologies*, volume 2 of *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford Science Publications, Clarendon Press, Oxford, England, 1993.

[66] Dov M. Gabbay, C.J. Hogger, J.A. Robinson, and D. Nute, editors. *Logical Foundations*, volume 1 of *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford Science Publications, Clarendon Press, Oxford, England, 1993.

[67] Dov M. Gabbay, C.J. Hogger, J.A. Robinson, and D. Nute, editors. *Nonmonotonic Reasoning and Uncertain Reasoning*, volume 3 of *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford Science Publications, Clarendon Press, Oxford, England, 1994.

[68] Dov M. Gabbay, C.J. Hogger, J.A. Robinson, and D. Nute, editors. *Epistemic and Tempral Reasoning*, volume 4 of *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford Science Publications, Clarendon Press, Oxford, England, 1995.

[69] D. Garlan. Research directions in software architecture. *ACM Computing Surveys*, 27(2):257–261, June 1995. .

[70] D. Garlan. Formal approaches to software architecture. In *Studies of Software Design. ICSE '93 Workshop. Selected Papers*, pages 64–76, Berlin, Germany, 1996. Springer-Verlag. .

[71] D. Garlan and M. Shaw. Experience with a course on architectures for software systems. In *Software Engineering Education. SEI Conference 1992; San Diego, CA, USA*, pages 23–43, Berlin, Germany, 199. Springer-Verlag. .

[72] D. Garlan and M. Shaw. *An introduction to software architecture*, pages 1–39. World Scientific, Singapore, 1993. .

[73] J.A. Goguen and LuQi. Formal methods and social context in software development. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *TAPSOFT '95: Theory and Practice of Software Development. 6th International Joint Conference CAA/FASE*, pages 62–81, Aarhus, Denmark, 1995. .

[74] Jan Goossenaerts and Dines Bjørner. An Information Technology Framework for Lean/Agile Supply-based Industries in Developing Countries. Technical Report 30, UNU/IIST, P.O.Box 3058, Macau, 1994. ublished in *Proceedings of the International Dedicated Conference on Lean/Agile Manufacturing in the Automotive Industries*, ISATA, London, UK.

[75] Jan Goossenaerts and Dines Bjørner. Interflow Systems for Manufacturing: Concepts and a Construction. Technical Report 31, UNU/IIST, P.O.Box 3058, Macau, 1994. ublished in *Proceedings of the European Workshop on Integrated Manufacturing Systems Engineering.*

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

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

[78] M. Gruninger and M.S. Fox. The Logic of Enterprise Modelling. In *Modelling and Methodologies for Enterprise Integration, see [13]*, pages 141–157, November 1995.

[79] Nicola Guarino. Formal Ontology, Conceptual Analysis and Knowledge Representation. *Intl. Journal of Human–Computer Studies*, (43):625–640, 1995.

[80] Nicola Guarino. Some Organising Principles for a Unified Top–level Ontology. Int.Rept. 02/97, v2.1, Italian National Research Council (CNR), LADSEB–CNR, Corso Stati Uniti 4, I–35127 Padova, Italy. guarino@ladseb.pd.cnr.it, 1997.

[81] Carl Gunther. Higher Order Logics in Requirements Engineering. In Manfred Broay, editor, *Requirements Targeting Software (Systems) Engineering*, 1997.

[82] J. Guttag, J.J. Horning, and J.M. Wing. Larch in Five Easy Pieces. Technical Report 5, DEC SRC, Dig. Equipm. Corp. Syst. Res. Ctr., Palo Alto, California, USA, 1985.

[83] John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, , and J.M. Wing. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, Springer-Verlag New York, Inc., Attn: J. Jeng, 175 Fifth Avenue, New York, NY 10010-7858, USA, 1993.

[84] Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, 1990.

[85] P.A.V. Hall, D. Bjørner, and Z. Mikolajuk. Decision Support Systems for Sustainable Development: Experience and Potential — a Position Paper. Administrative Report 80, UNU/IIST, P.O.Box 3058, Macau, August 1996.

[86] Ian J. Hayes, editor. *Specification Case Studies.* International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1987.

[87] E. Haywood and P. Dart. Analysis of software system requirements models. In *Australian Software Engineering Conference*, pages 131–138. IEEE Comput. Soc. Press, 1996. .

[88] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw. Automated Consistency Checking of Requirements Specifications. *TOSEM: ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July, 1996. .

[89] Hennesy and Patterson. *Computer Architectures.* Addison Wesley, 199?

[90] Hennesy and Patterson. *Computer Organisations.* Addison Wesley, 199?

[91] J. Hoare, J. Dick, D. Neilson, and I.H. Sorensen. Applying the B technologies to CICS. In C. Gaudel M and J. Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods. Third International Symposium of Formal Methods Europe. Proceedings; Springer-Verlag, LNCS.* Formal Methods Europe, 1996. .

[92] Dang Van Hung and Zhou Chaochen. Probabilistic Duration Calculus for Continuous Time. Research Report 25, UNU/IIST, P.O.Box 3058, Macau, May 1994. resented at *NSL'94 (Workshop on Non-standard Logics and Logical Aspects of Computer Science, Kanazawa, Japan, December 5–8, 1994)*, submitted to *Formal Aspects of Computing.*

[93] Dang Van Hung and Phan Hong Giang. A Sampling Semantics of Duration Calculus. Research Report 50, UNU/IIST, P.O.Box 3058, Macau, November 1995. ublished in: *Formal Techniques for Real-Time and Fault Tolerant Systems*, Bengt Jonsson and Joachim Parrow (Eds), LNCS 1135, Spriger-Verlag, pp. 188–207, 1996.

[94] IEEE. *Proceedings of the Second International Conference on Requirements Engineering;C olorado Springs, CO, USA*, Los Alamitos, CA, USA, 1996. IEEE Comput. Soc. Press.

[95] Daniel Jackson. Structuring Z Specifications with Views. *ACM Transactions on Software Engineering and Methodology*, 4(4):365–389, October 1995.

[96] M. Jackson. *System Design.* Prentice-Hall International, 1985.

[97] M. Jackson. Problems and requirements (software development). In *Second IEEE International Symposium on Requirements Engineering (Cat. No.95TH8040)*, pages 2–8. IEEE Comput. Soc. Press, 1995. .

[98] Michael Jackson. Problems, methods and specialisation. *Software Engineering Journal*, pages 249–255, November 1994. .

[99] 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.

[100] Michael A. Jackson. *Software Development Method*, chapter 13, pages 215–234. Prentice Hall Intl., 1994. Festschrift for C. A. R. Hoare: *A Classical Mind*, Ed. W. Roscoe.

[101] T. Janowski and C.M. Acebedo. Virtual Enterprise: On Refinement Towards an ODP Architecture. Research Report 69, UNU/IIST, P.O.Box 3058, Macau, May 1996. .

[102] Tomasz Janowski. Domain Analysis for Manufacturing: Formalization of the Market. Research Report 63, UNU/IIST, P.O.Box 3058, Macau, March 1996. .

[103] Tomasz Janowski and Rumel V. Atienza. A Formal Model For Competing Enterprises, Applied to Marketing Decision-Making. Research Report 92, UNU/IIST, P.O.Box 3058, Macau, January 1997. .

[104] C.B. Jones. *Systematic Software Development using VDM.* Prentice Hall International, second edition, 1990.

[105] M. Kirikova and J.A. Bubenko Jr. Software requirements acquisition through enterprise modelling. In *SEKE '94. The 6th International Conference on Software Engineering and Knowledge Engineering*, pages 20–27, Skokie, IL, USA, 1994. Knowledge Syst. Inst. .

[106] G. Kosters, H.-W. Six, and J. Voss. Combined analysis of user interface and domain requirements. In *Second International Conference on Requirements Engineering (Cat. No.96TB100037); Colorado Springs, CO, USA*, pages 199–207. IEEE Comput. Soc. Press, 1996. .

[107] Souleymane Koussoubé. Knowledge-Based Systems: Formalisation and Applications to Insurance. Research Report 108, UNU/IIST, P.O.Box 3058, Macau, May 1997.

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

[109] Peter Gorm Larsen. The IFAD VDM-SL Toolbox Brochures. Technical report, Institute for Applied Datalogy, Forskerparken 10, DK-5230 Odense M, Denmark, 1994.

[110] Nancy G. Leveson, M.P.E. Heimdahl, H H. Hildreth, and J.D. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994. .

[111] P. Loucopoulos and E. Kavakli. Enterprise modelling and the teleological approach to requirements engineering. *International Journal of Intelligent & Cooperative Information Systems*, 4(1):45–79, March 1995. .

[112] M et al. M. Jarke. Requirements engineering: an integrated view of representation, process, and domain. In I. Sommerville and M. Paul, editors, *Software Engineering - ESEC '93. 4th European Software Engineering Conference Proceedings; Garmisch-Partenkirchen, Germany*, pages 100–114, Berlin, Germany, 1993. Springer-Verlag. .

[113] N.A.M. Maiden and A.G. Sutcliffe. Requirements critiquing using domain abstractions. In *First International Conference on Requirements Engineering (Cat. No.94TH0613-0); Colorado Springs, CO, USA*, pages 184–193, Los Alamitos, CA, USA, 1994. IEEE Comput. Soc. Press. .

[114] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive Systems: Specifications.* Addison Wesley, 1991.

[115] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive Systems: Safety.* Addison Wesley, 1995.

[116] T.L. McCluskey, J.M. Porteous, Y. Naik, C.N. Taylor, and S. Jones. A requirements capture method and its use in an air traffic control application. *Software - Practice and Experience*, 25(1):47–71, January 1995. .

[117] John A. McDermid. Special section on Z. *Software Engineering Journal*, 4(1):25–72, January 1989.

[118] M. Moulding and L.Smith. Combining formal specification and CORE: an experimental investigation. *Software Engineering Journal*, 10(2):31–42, March 1995. .

[119] Nikolaj Nikitchenko. Towards Foundations of the General Theory of Transport Domains. Research Report 88, UNU/IIST, P.O.Box 3058, Macau, December 1996. .

[120] Jørgen Fischer Nilsson. An Algebraic Logic for Concept Structures. In H. Jaakkola et al., editor, *Information Modelling and Knowledge Bases*, Amsterdam, The Netherlands, 1993. IOS Press.

[121] Jørgen Fischer Nilsson. Object Logic for Conceptual Modelling — with a Medical Domain as Case Study. In Y. Tanaka et al., editor, *Information Modelling and Knowledge Bases*, pages 330–343, Amsterdam, The Netherlands, 1996. IOS Press.

[122] Jørgen Fischer Nilsson and Jari Palomäki. A Categorical View on Concept Structures. In H. Kangassolo et al., editor, *Information Modelling and Knowledge Bases*, pages Chapter 16, 239–256, Amsterdam, The Netherlands, 1995. IOS Press.

[123] Roger Noussi. An Efficient Construction of a Domain Theory for Resources Management: A Case Study. Research Report 107, UNU/IIST, P.O.Box 3058, Macau, May 1997.

[124] B. Nuseibeh, J. Kramer, and A. Finkelstein. Expressing the relationships between multiple views in requirements specification. In *15th International Conference on Software Engineering (Cat. No.93CH3270-6)*, pages 187–96, Los Alamitos, CA, USA, 1993. EEE Comput. Soc. Press. .

[125] J.T. Nutter. Epistemology. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*. John Wiley, London, UK, 1997.

[126] R. Reiter. *The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression*, chapter in "Artificial Intelligence and Mathematical Theory of Computation: Papers in Honour of John McCarthy". Academic Press, San Diego, 1991.

[127] C. Shekaran, D. Garlan, and et al. The role of software architecture in requirements engineering. In *First International Conference on Requirements Engineering (Cat. No.94TH0613-0); Colorado Springs, CO, USA*, pages 239–245, Los Alamitos, CA, USA, 1994. IEEE Comput. Soc. Press. The role of software architecture (which reflects high-level implementation constraints) in requirements engineering is clarified by providing perspectives on relevant issues, including the following: is requirements engineering merely a front end to the software development process that is concerned only with problem definition? Is software architecture an application-specific, high-level design of a system (for example, "an object-oriented system with a specified object hierarchy")? What is the relationship between the problem definition and the solution structure? What is the relationship between the roles of requirements engineer, software architect, and application domain specialist?. .

[128] E.V. Sørensen, J. Nordahl, and N.H. Hansen. From CSP Models to Markov Models: A Case Study. *To be published in IEEE Transactions on Software Engineering*, Dept. of Computer Science, Technical University of Denmark, August 15 1991.

[129] J. Michael Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, UK, January 1988.

[130] J. Michael Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, UK, January 1988.

[131] J. Michael Spivey. An Introduction to Z and Formal Specifications. *Software Engineering Journal*, 4(1), January 1989.

[132] J. Michael Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1989.

[133] J. Michael Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK, 1989.

[134] J.J.P. Tsai, T. Weigert, and H.C. Jang. A hybrid knowledge representation as a basis of requirement specification and reasoning. In *2nd International IEEE Conference on Tools for Artificial Intelligence (Cat. No.90CH2915-7)*, pages 70–76. IEEE Comput. Soc. Press, 1990. .

[135] Jim C.P. Woodcock. Teaching How to Use Mathematics for Large-Scale Software Development. *Bull. BCS-FACS*, July 1988.

[136] Jim C.P. Woodcock. Calculating Properties of Z Specifications. *ACM SIGSOFT Software Engineering Notes*, 14(4):43–54, 1989.

[137] Jim C.P. Woodcock. Structuring Specifications in Z. *Software Engineering Journal*, 4(1):51–66, January 1989.

[138] Jim C.P. Woodcock. *Using Z – Specification, Refinement and Proof*. Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, 1991. In preparation.

[139] Jim C.P. Woodcock and Martin Loomes. *Software Engineering Mathematics: Formal Methods Demystified*. Pitman Publishing Ltd., London, UK, 1988.

[140] Tan Xinming. Enquiring about Bus Transport–Formal Development using RAISE. Technical Report 83, UNU/IIST, P.O.Box 3058, Macau, September 1996. .

[141] S. Yamamoto, H. Tadaumi, and M. Ueno. DREM: domain-based requirements engineering methodology. *NTT R & D*, 45(8):711–718, 1996. n Japanese. .

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

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

[144] Liu ZhiMing, A.P. Ravn, E.V. Sørensen, and Zhou ChaoChen. A Probabilistic Duration Calculus. Technical report, Dept. of Computer Science, Technical University of Denmark, February 1992. Submitted for: 2nd Intl. Workshop on Responsive Systems, Japan, 1992.

Index

Rapid Prototyping and Incremental Evolution

David A. Dampier
National Defense University
Information Resources Management College
Building 62, 300 5th Avenue
Fort Lesley J. McNair, DC 20319
Fehler! Textmarke nicht definiert.

Abstract

Software development is no longer an enterprise where the traditional waterfall method of system construction is acceptable. Information technology is changing at a pace that requires complete system development and fielding in less than 18 months. This is due in part to faster technology insertion, and in part by increased user expectations. Both reasons provide justification for changing the way software is built and fielded. Increased user expectations require that we involve the user more in the requirements engineering process, and deliver the software to the user much more quickly. Faster technology insertion requires that we incorporate new technology into existing products much faster and with less rework.

A new software *evolution* paradigm is needed to accomplish these goals, along with the automated tools to realize the benefits. Computer-Aided prototyping is one such method that incorporates the goals and opinions of the user from the beginning of the software evolution process, throughout the lifecycle, and into retirement. Automated tools, like the Computer-Aided Prototyping System [1], assist the software developer in building executable prototypes of a software system very quickly, involving the user in an iterative build, execute, modify loop until the user is satisfied with the demonstration of the prototype. The prototype is then used to build the final version of the software through the use of the architecture included in the prototype, as well as the validated set of requirements constructed during the prototyping process. This final version is delivered very quickly, hopefully before the user's requirements have an opportunity to change.

In the event that the user's requirements do change, new requirements can be incorporated into a next version of the system by using the same iterative process where the fielded version of the system provides the base version of the process. This incremental evolution process can proceed throughout the life of the system.

What is needed in the paradigm is a method for automating the parts of the process that are not already automated by CAPS. These include computer-aided construction of the prototype through intelligent interpretation of requirements into design, and better mechanisms for finding and retrieving reusable components from a repository. Current capability in CAPS provides for the ability to retrieve reusable components from a stand-alone repository built for the purpose, but to be useful in general, a methodology that uses some commercial standard, such as CORBA, for storing the components is needed to allow distributed access to multiple repositories. Additionally, an integrated automated testing capability is needed to provide for more robust

prototypes to be delivered to customers as increments of the final system. Current prototypes are not industrial strength, and therefore cannot be expected to perform in a safe manner in the user's environment.

This idea of a new paradigm to build software is not new. Many have tried to develop new ways to do the same things. I do not propose a revolutionary new way o do software development, but merely propose a new way to use some existing technology to satisfy a growing need, quicker delivery of software products that can be maintained more easily, and updated more rapidly.

1. Luqi and Ketabchi, M., "A Computer-Aided Prototyping System" *IEEE Software*, March 1988.

Combining and distributing hierarchical systems

Chris George[1] and Đỗ Tiến Dũng[2]

[1] United Nations University International Institute for Software Technology, Macau
[2] Ministry of Finance, Hanoi, Vietnam

Abstract. It is possible with RAISE to specify and do most refinement in an applicative framework, and then transform the concrete applicative specification into an imperative sequential or concurrent one. This transformation changes from a style more appropriate to proof of refinement to a style more appropriate to implementation.
The resulting imperative specification is typically hierarchical, with upper levels calling the functions of lower ones. This paper presents a further stage of development in which the hierarchical structure is transformed into a distributed one, and components communicate asynchronously. This also allows "horizontal" communication between components of previously separate hierarchies.
A major design aim is to reuse the hierarchical specification, as far as possible extending the existing modules by standard, generic components. The method should achieve correctness by construction, and be amenable to quality control; it is is an example of an engineering approach using standard components and standard assembly techniques.
The method is illustrated by collaborative work done between UNU/IIST and the Vietnamese Ministry of Finance in developing a specification of a national financial information system.
Keywords Formal specification, development, refinement, reuse, restructuring, distributed systems, software engineering

1 Introduction

We take it that engineering, as opposed to science, creates artifacts as far as possible through combining existing components. Speed and cost are minimised and reliability maximised through having to invent from fresh as little as possible. In this process the engineer exploits the known properties of the components, and the known laws of the combining activity which allows the engineer to compute the properties of the combinations.

In this paper we describe the development of a distributed system, a financial information system, by developing first an applicative (or functional) specification and then transforming this, first into an imperative concurrent but still hierarchical system, and then into a distributed system. The first of these transformations follows the existing ideas of the RAISE method [1]. The second transformation is new and is based on a small number of standard components. Hence it exemplifies an essentially engineering approach.

In section 2 we describe the problem we tackled, the development of a specification of a national financial information system for Vietnam. In section 3 we describe the RAISE method in outline and show how it was applied to the problem. In section 4 we describe

how the transformation to a distributed system was achieved. Section 5 is a concluding discussion.

2 A Financial Information System

During 1996–7 United Nations University International Institute for Software Technology (UNU/IIST) in Macau and the Vietnam Ministry of Finance (MoF) undertook a joint project called MoFIT (Ministry of Finance Information Technology) aimed at doing the domain analysis and specification for a national financial information system for Vietnam. The aim was to specify the major components of such a system and to specify the main activities and information flows. A second aim was to train software engineers from Vietnam in the relevant techniques. As well as the first author from UNU/IIST, the project involved seven mainly young software engineers from Vietnam: four from the MoF, one from the Institute of Information Technology in Hanoi and one from Hanoi University. During the 16 months of the project these people each spent between 6 and 12 months working at UNU/IIST. As well as the main work described here studies were also made of other aspects like system security and the possible effects of changes in taxation policy. The results are described in two UNU/IIST technical reports [2, 3] which in turn reference a number of more detailed project reports.

Vietnam is divided into 61 provinces, provinces are divided into districts, and districts into communes. The major government ministries reflect this structure, with offices at the national, province, district and in some cases commune levels. So much of the collection or dissemination of information follows this hierarchical structure. In collecting information about taxes, for example, districts will supply information to their provincial offices, which will merge and perhaps summarise it and send it to the national office for the final merge into national information. Changes in taxation policy, or requests for information, flow down the hierarchy in the obvious manner.

The main organisation concerned with generating revenue is the taxation system, which is part of the MoF. In the first phase of the project all the engineers had experience of developing software for this system, mainly packages for particular tasks for province and district taxation offices. So in this phase we concentrated on analysing and specifying the taxation system.

In the second phase we considered other components. The treasury system is concerned with the actual collection and disbursement of money, with offices at national, provincial and district levels. The budget system is concerned with collecting budgetary estimates at the commune, district, province and national levels and, after government decision on the final figures, distributing actual annual budgets at the various levels and then monitoring these budgets. We also looked at two systems which exist only at the national level: the external loans and external aid systems.

2.1 The Taxation System

Taxation in Vietnam is currently primarily on enterprises. There are various categories of tax, such as profit taxes and sales taxes that may be levied. Provincial tax departments are responsible for larger enterprises as well as their district offices; district offices are concerned with smaller enterprises.

Province and district taxation offices therefore share the task of demanding, collecting and accounting for taxes. They need to maintain for each taxpayer

- a *roll* of comparatively static information about the taxpayer, including basic details like name and address as well as information about the kind of business the taxpayer is in, from which the applicable categories of tax can be determined
- *bases* or figures collected from taxpayers about actual turnover, profits, etc.
- *accounts* recording taxes demanded, paid and owing for each category of tax for each taxpayer in each period.

They also need the current national taxation rules, called the *regime*, for tax calculation.

National and provincial taxation offices share the tasks of collecting, merging and summarising reports from their constituent offices at the immediately lower level.

It is apparent that there are several functional or organisational components of the taxation system that one would like the design to reflect:

- accounting for each category and period for each taxpayer
- registration of taxpayers
- recording base information for taxpayers
- making, merging and summarising reports
- structurally relating districts to provinces, provinces to the national office

Making the structure of the specification reflect the main conceptual components aids in the comprehension of the overall system. Making the components separate with the standard properties of internal coherence and minimal linkage makes them easier to develop independently and robust against changes to other components.

One would also like to specify only once shared data structures and functions over them.

2.2 The taxation system specification

The taxation system specification does meet these structural requirements. There are separate components for a regime, for registration, for a roll, for a base and the tax calculation from it, for a collection of bases, for an account and a collection of accounts. These combine as illustrated in figure 1 into a *group*, which provides all the functions for dealing with a collection of taxpayers. It also shares some other specifications that will be used globally: an abstract description of a report format with a functions to merge and summarise reports with common formats, and an abstract summation function that can be applied to the range of a mapping.

In figure 1 nested boxes indicate extension (inheritance), continuous lines indicate module dependency where the lower box is used to make an object in the upper one, and broken lines indicate dependency on shared modules through parameterisation.

The main structure of the specification then follows the hierarchical structure of the taxation system. Each district and province has a group; a province has a number of districts and the national or *general* taxation department (GTD) has a collection of provinces. See figure 2. For example, the type *Office* at the provincial (PTD) level is defined as a record containing its group and its district offices represented as a mapping from their identifiers to their offices:

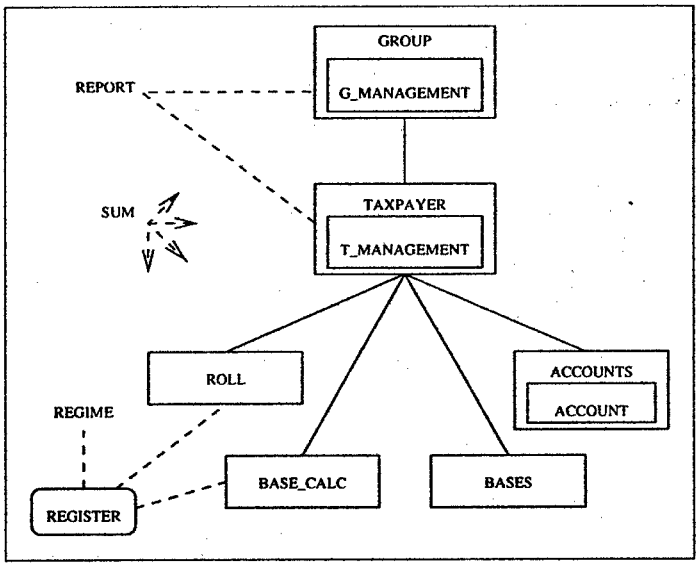


Fig. 1. Modules involved in specifying a taxation group

type
  Office :: (1)
    taxpayers : GR.Group
    offices : T.DTDid $\overrightarrow{m}$ DTD.Office

The prefixes in the type names indicate that the types are defined in other modules.

There is a function *mk_report* that specifies that a provincial report with a particular format is the result of collecting the district reports, merging these, creating the report from its own group, and merging this and the merged report from the districts. Thus the basic requirement that a provincial report combines these elements is clearly specified.

The specification at this point consists of about 1000 lines of RSL in 22 modules. The specification is applicative (functional). We wanted to develop it further towards a possible implementation. We had taken some account of a possible implementation strategy in that the main component of the Group module was a database (specified abstractly as a standard generic module) instantiated with a structure of information to be recorded about each taxpayer. So we separated the storage and retrieval of information about taxpayers from its processing, and provided a basis for a database implementation. Further design work would suggest possible detailed database schemas or relations that could be used for implementation. But such an implementation would be imperative, not applicative. Additionally, the actual system runs asynchronously, with different offices separated geographically and within one office probably several concurrent users. Finally,

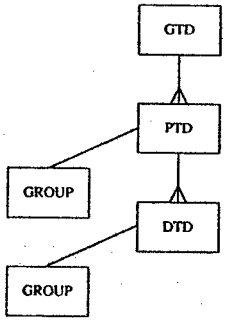


Fig. 2. Taxation system hierarchy

as we shall see, the taxation system communicates with other systems like the treasury and budget systems.

To see how we developed the specification to reflect these issues we first describe the RAISE method in more general terms.

3 The RAISE Development Method

The RAISE specification language (RSL) [4] allows specification in both applicative and imperative styles, and of both sequential and concurrent systems. The applicative/imperative and sequential/concurrent distinctions are orthogonal, giving four possible styles, but the applicative concurrent style is rarely used. So we use applicative, imperative and concurrent as abbreviations for applicative sequential, imperative sequential and imperative concurrent respectively.

RSL supports the specification of data types in the standard algebraic style, by defining abstract types (sorts) and axioms over their generators and observers.

A design goal of RSL was uniformity, and so it is also possible to specify imperative programs using axioms, and also to specify concurrent systems in the same way. This allows equational reasoning about all styles of specification.

When one speaks about an abstract data type, one is being abstract about the structure of the type. The imperative counterpart is to be abstract about the variable(s) involved (in the programming language sense of a variable as an assignable entity whose contents can later be retrieved). One can see the collection of variables (with their types) of an imperative specification as corresponding to the "type of interest" [5] of an applicative specification.

The concurrent part of RSL is based on process algebra (similar to CSP [6] and CCS [7]) with communication of values along channels. One can abstract away from the channels involved (and hence also about the possible internal communications with sub-processes).

Consider a simple example of an abstract data type with a generator *empty* and an observer *is_empty*. Here are the appropriate axioms in the three styles:

[applicative]
  is_empty(empty) ≡ **true**

[imperative]
  empty() ; is_empty() ≡ empty() ; **true**

[concurrent]
  ∀ test : **Bool** $\xrightarrow{\sim}$ **Unit** •
    (main() ‖ empty()) ‖ test(is_empty()) ≡ (main() ‖ empty()) ‖ test(**true**)

The strong equivalence "≡" between expressions compares not only results but also effects, i.e. changes to variables and communications on channels. (In the applicative case it could be replaced by "=", as there are no effects.) The applicative generator *empty* becomes in the imperative case a function of the same name that will change some variables so that the current state is "empty". The applicative observer *is_empty* becomes an imperative function that can read some or all of these variables. Thus the imperative axiom says that performing *empty* followed by performing *is_empty* is in every way equivalent to performing *empty* and returning **true**.

In the concurrent case we need a *main* or server process that controls the imperative state. The functions *empty* and *is_empty* become "interface processes" that interact with the server to change or interrogate its internal state. (In object oriented terminology these would be called "methods".) The interlock operator "‖" is like the parallel operator but allows its constituent processes to communicate only with each other until one of them terminates. The *test* process is just a technique necessitated by interlock requiring its arguments to be of type **Unit**. So we can read the concurrent axiom as saying that if we force the server *main* to communicate with *empty*, and the resulting process with *is_empty*, the result will be in every way equivalent to forcing the communication with *empty* and obtaining **true**.

It should be clear that all these axioms say essentially the same thing. If you make it empty, and then ask if it is, it will be. But the effort and machinery to make this simple assertion becomes progressively more difficult as we proceed to imperative and then concurrent styles. This was certainly the experience of early users of RSL.

The difference does not only apply to specification, but also to reasoning about specifications. Whether this is done manually or with a proof tool, our experience is that proving the "same" property in the different styles for the "same" specification involves effort and difficulty on a ratio of something like 1:2:5 for the three styles. These figures are only impressions — we have made no measurements. We only want to make the point that things get much more difficult. A proof tool with better strategies could undoubtedly alleviate the problems, but we doubt the disparity can be removed.

So we conclude that (abstract) applicative specifications are easiest to construct and to reason about, but imperative or concurrent systems are what we typically need to implement. There does seem to be a notion of them being the "same" thing in some sense — or at least there being imperative and concurrent counterparts to applicative specifications. So perhaps we can transform the latter into the former. If we can supply a notion of correctness, i.e. define precisely what we mean by "same", then we have a possible development method.

This was described in the book on the RAISE method [1]. We can follow the development route illustrated in figure 3. We start with a more or less abstract applicative specification. Making this concrete essentially involves making its types concrete and defining the required functions over these concrete types. Showing refinement involves showing that these functions satisfy the axioms of the more abstract specifications. The concrete types typically are records or products of other types which sometimes merit being made the "types of interest" of subsidiary modules. This process naturally produces a hierarchy of modules (with some complications when we use parameterisation to make modules generic or to allow them to be "shared").

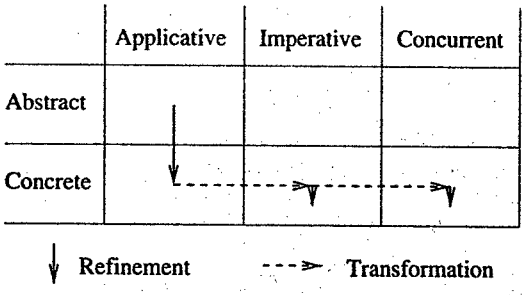


Fig. 3. Development route

When we have a concrete applicative specification we can transform it into a concrete imperative one, using a standard set of transformation rules. This transformation operates on a module by module basis, and preserves the structure of the specification. We arrive at a similarly hierarchic specification where the "leaf" modules have variables and functions to change or report on their state. "Branch" modules normally have no variables; their functions call the functions of the leaf modules below them.

Finally there may be some small refinement steps that are best done in an imperative context, like replacing recursion with iteration, or to make the specification translatable into a programming language, like introducing iteration to refine existentially or universally quantified expressions.

The syntax of the transformation is straightforward, but what about its semantics? What is the semantic relation between the applicative specification and the imperative one generated from it?

The imperative specification cannot be a refinement of the applicative one, because the signatures of the functions have changed: parameters corresponding to state variables have disappeared. The refinement relation in RSL is required to allow substitution in a system of a component by a refinement of it. So certainly such changes in signature are precluded. But there is a meta-theorem (figure 4) that says

- there is an abstract imperative specification *I0* (i.e. one with no explicitly defined variables) that refines the concrete imperative one **I1**

– a conservative extension $I0 + D$ of this abstract imperative specification refines the original abstract imperative one **A0**

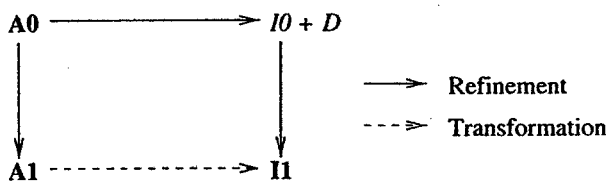


Fig. 4. Transformation theorem

This construction was originally described in [8].

The point is that there is no need to write the abstract imperative specification *I0* or the extension *D*: we know they exist and we know that the concrete imperative specification constructed by transformation is "correct" with respect to the applicative one.

Another way to see this notion of correctness is to consider again the three axioms (1) of section 2.2. The transformation ensures that if the applicative specification has a property, such as that relating *empty* and *is_empty*, then the imperative specification will have the corresponding imperative property, where "corresponding" is defined by the transformation.

Another point to note is that the method need not even start with an abstract applicative specification. It is possible to start with a concrete one, which is for most people the easiest starting point, and there is a simple abstraction method that will create an abstract applicative one from it. In the MoFIT project we did almost no refinement; most of the applicative modules have concrete types and explicit algorithms.

A further transformation, first described in [1], will produce a concrete concurrent specification from the concrete imperative one. This again applies module by module and maintains the overall structure of the specification. Leaf modules contain the imperative state components embedded in "server" processes.

As an example to illustrate these ideas we consider the buffer that we will use later both for the message system and the in-tray of our distributed system. We start with a parameter scheme *BUFF_PARM* which postulates a type *Elem*, a particular *null* value of this type, and a test *is_null*:

```
scheme BUFF_PARM =
class
  type Elem
  value
    null : Elem • is_null(null),
    is_null : Elem → Bool
end
```

The applicative specification *A_BUFFER* given here is concrete in that its type of interest *Buffer* is concrete: it is a list of *Elem* values. The function *put* is explicitly specified: it returns a new buffer with the extra value appended. The *get* function is specified implicitly. It takes a predicate as a parameter allowing it to be used either as a function to get the first element in the buffer (by making the predicate "λ x:X.Elem • **true**") or for extracting an element with some particular property. We will need this feature for the in-tray later, when we need to be able to extract a message with a particular number. Failure, because the buffer is empty or there is no element with the required property, is indicated by returning the *null* element.

```
scheme A_BUFFER(X : BUFF_PARM) =
class
  type Buffer = X.Elem*
  value
    put : X.Elem × Buffer → Buffer
    put(e, b) ≡ b ⌢ ⟨e⟩,

    get : (X.Elem → Bool) × Buffer → X.Elem × Buffer
    get(f, b) as (e, b′) post
      (∀ x : X.Elem • x ∈ elems b ⇒ ∼ f(x)) ∧ e = X.null ∧ b′ = b
      ∨
      (∃ b1, b2 : Buffer •
        b = b1 ⌢ ⟨e⟩ ⌢ b2 ∧
        b′ = b1 ⌢ b2 ∧ f(e) ∧
        (∀ x : X.Elem • x ∈ elems b1 ⇒ ∼ f(x)))
end
```

One way to transform the applicative *A_BUFFER* to a concurrent *C_BUFFER* is to use the former in the definition of the latter:

```
scheme C_BUFFER(X : BUFF_PARM) =
hide A, buff, put_ch, get_ch, get_res_ch in
class
  object A : A_BUFFER(X)
  variable buff : A.Buffer := ⟨⟩
  channel put_ch, get_res_ch : X.Elem, get_ch : X.Elem → Bool
  value
    main : Unit → in put_ch, get_ch out get_res_ch write buff Unit
    main() ≡
      while true do
        buff := A.put(put_ch?, buff)
        []
        let (e, b′) = A.get(get_ch?, buff) in
          buff := b′ ; get_res_ch!e
        end
      end,
```

```
    get : (X.Elem → Bool) → in get_res_ch out get_ch X.Elem
    get(f) ≡ get_ch!f ; get_res_ch?,

    put : X.Elem → out put_ch Unit
    put(e) ≡ put_ch!e
end
```

Here we have a server process *main* that runs for ever, mostly waiting for interactions with the "interface processes" *put* and *get*. The imperative state is held in a variable *buff*, and three channels are used for communication. The variable and channels are hidden, so the only possible interactions are via the interface processes. For example, when *get* is called with actual parameter a predicate *f*, it will communicate with *main* by outputting *f* on the channel *get_ch*. *main* uses the functions defined earlier in *A_BUFFER* to compute the result from the predicate *f* and the current value of the variable *buff*. Then the new buffer value is assigned to *buff* and the result element output back to *get* on the *get_res_ch* channel. *get* then terminates, returning the element value it received. It should be intuitively clear that the concurrent buffer "behaves like" the applicative one, and this can be formalised in terms of transforming applicative properties into concurrent ones.

When the types of interest of sequential modules are functions or mappings over finite types, the concurrent system has an extra level through there being RSL "object arrays". Thus the mapping in the provincial tax department of district tax department identifiers to district tax department offices results in an array of objects modelling district tax offices.

We can see the effect of this transformation for our provincial tax office. The type definition (1) of section 2.2 becomes

```
object
  GR : GROUP,
  DTDS[ id : T.DTDid ] : DTD
```

We have an object for the province's group of taxpayers and an array of objects representing its constituent district tax departments.

The semantic relation, and hence the notion of correctness of the concurrent system, is similar to the imperative case. There will exist an abstraction from the concrete concurrent specification to an abstract concurrent one, a conservative extension of which can be shown to implement the original abstract applicative one. We can guarantee that if the applicative specification has a property, the concurrent one will have the corresponding transformed property.

The concurrent architecture has some convenient features. In particular:

- It is guaranteed to be deadlock free.
- The states of the imperative components are independent, since all communication is between leaf and branch nodes. This means in turn that it is possible for branch modules to call the interface processes of their leaf nodes in parallel instead of sequentially, with the same results; there is no interference.

It seems possible in practice to further develop this calling structure, in particular to deal with "shared" nodes, so that leaves can call the interface processes of other leaves. This

requires care in the proper sequencing of calls in branch modules, and also requires that the dependencies between modules are acyclic.

It is not suggested that all concurrent systems can be designed in this particular way. But it does seem to apply quite conveniently to many systems and to give a very satisfactory architecture.

4 From Concurrent to Distributed

The construction described in the previous section was applied to the taxation specification. In the meantime we had worked on the treasury and budget systems. These exhibit essentially the same kind of hierarchic structure based on provinces, districts (and, for the budget system, communes). The three specifications needed to be combined into a single specification, since in practice the three systems communicate, and they do so at the local level. For example, when someone pays tax they actually pay it at the local treasury office. This sends a notification to the corresponding tax office. It also reports amounts collected to the local budget office, since this monitors budget performance. We had modelled for each system the receipt of such information but not (since we had three separate specifications) the transmission of it.

We also wanted to include the two other specifications we had done, of the external loans and external aid systems. These do not exhibit the same problems since they only have national offices.

The structure of the hierarchic specifications seemed wrong for introducing "horizontal" communication between local offices of different systems. The provincial offices are modelled by an array of objects "inside" the national one, and the district offices are similarly "inside" the provincial ones. This is really just a conceptual issue; there is no reason why a treasury office in district D of province P should not call a function named *Tax.P.D.pay* to report a tax payment, but to some members of the team it seemed wrong to apparently pass the call through the national office.

More real is the problem that communication in the hierarchic systems is synchronous, while communication in the actual systems will be asynchronous. The means of communication between offices vary, and at present very few are electronic. Asynchrony applies vertically as well as horizontally — and often the delays are longest in this direction. District treasury and taxation offices may be co-located, but some way from their provincial offices.

So we wanted to move from the situation illustrated in figure 5 to that illustrated in figure 6.

4.1 Construction

There are two issues: how to construct the combined and distributed system, and its semantics: how to relate its properties to those of the separate hierarchical ones. The first we describe in this section 4.1, the second in section 4.2.

The aim was to achieve this restructuring of the specification while reusing as much as possible of the work done already. It was clear that the restructuring would need some additional components — the message system for a start. We wanted to make these components as far as possible generic and hence reusable between the systems we had and also for similar problems in the future.

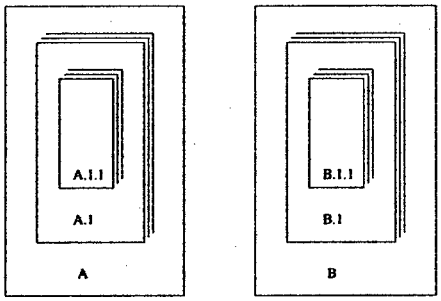


Fig. 5. Separate hierarchical systems

The Message System The message system clearly needs a universal set of addresses for offices in different systems. This is easy enough to specify. A district address, for example, has the form *System.Province.District* which makes it easy for a district treasury office, for example, to address a message to its corresponding tax office, or to its treasury provincial office.

We also apparently need a universal message type. We did not want to enforce such a type across the component systems, so instead we specified a message type for each system plus a global one, together with *encode* and *decode* functions within each system between its type and the global one. There are axioms for each system

$$\forall\ m : \text{System\_message} \bullet \text{decode}(\text{encode}(m)) = m$$

to guarantee correct message passing within the system. This leaves open the design of suitable functions to deal with the encoding and decoding of particular kinds of messages that pass between systems without it necessarily following that the tax system, say, can read all messages intended to stay within the treasury system.

The message system is specified as an array of the concurrent buffers we specified earlier, one buffer for each address. The *get* function for the array takes an address as argument and gets the next message in the buffer with that address. It does not use the feature we included in the buffer for extracting an element satisfying a predicate: we used that for the "in-tray" component described later in section 4.1.

We decided to enforce a rule that all messages are numbered (by defining a module to store and increment a number and instantiating it in each office, so that address and number together can uniquely identify a message), and include the sender's address, and a protocol that all messages are answered by at least an acknowledgement. The reply carries the same message number as the message it is answering or acknowledging. This, as we shall see, enables the automation of report collection and, if required, of other activities.

A "null" message may be the result of faulty communication, or merely the result of seeking a reply that has not yet arrived.

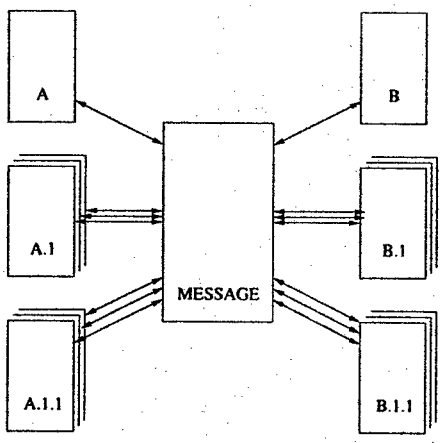


Fig. 6. Distributed and combined systems

An In-tray Each office has an in-tray for receiving incoming messages. This is an instantiation of the same buffer used to make the message system, but now we shall also use the facility for extracting a message by number, to allow for replies to messages to be extracted.

It is interesting to note that the function to extract a reply from an in-tray is a loop — it keeps trying until a non-null message is extracted. This breaks the normal design rule that only servers must potentially loop for ever: this is an interface process intended to interact with server processes, and must be guaranteed, assuming it finishes waiting to interact with a server, to terminate. In practice such a process will need an overall time-out or repetition limit to prevent it looping for ever, and also some delay between iterations to prevent "race" conditions. We could have specified this but it did not seem worth it — it is a problem easily solved at the final implementation stage — so we indicated the problem merely by a comment.

A Secretary Messages sent to the message system are placed in the appropriate destination buffer. They need to be transferred from there to the destination's in-tray. This is a traditional role for a secretary — to open the mail. The name of this module is quite intentional — in many tax, treasury, etc. offices in Vietnam this is currently a manual process and likely to be so for some time. In specifying this system we are not assuming that all of it will be implemented in software. Some instances of some components will be manual, and our specification then represents our assumptions about what will be done. In fact any specification component of a larger system has a dual role. To the other components that use it it describes what assumptions they may make about that component; to its implementor it describes what services must be provided.

We wrote two versions of the secretary module. An "unskilled" one merely transfers messages from the message system to the in-tray. A "skilled" one is supplied with a

function that decides if a particular (decoded) message can be "handled" by the secretary, in which case the appropriate function is applied and the reply encoded and dispatched to the sender of the original message. So, for example, the collection of reports within the tax system can be handled by the secretary modules since the tax "generate report" functions already exist. If desired, the recording of tax payments notified by messages from the treasury system can be done because the "pay" function in the taxation system already exists, and an acknowledgement can be sent. All that is required is to write the "handle" functions that call the appropriate existing functions according to the data in the message. This needs to be written for each office (or each class of office perhaps) but is a straightforward task.

Secretaries never handle replies to messages sent from their offices: they just place them in the in-tray. There are other components, like the stubs described in the next section, that will be waiting for them.

Stub modules Each provincial office, for example, contains in the hierarchical version an array of objects representing its district offices. These will no longer appear inside it, but need to be replaced by simple stub modules. For each function in the hierarchical system called in a lower object from a higher, the stub module will apparently provide the same function, returning the same result. But in fact it will send an appropriate message to the office for which it is a stub, and wait for a reply (recognised by message number) to appear in its office's in-tray. It then returns the content of the reply and thus (assuming no communication faults) appears, modulo some delay, to act exactly like the function it replaced.

Changes to existing modules It should be apparent that the only modules that need to be changed in forming the distributed system are those for each system defining the office at the national, provincial, district and possibly commune levels. Each needs to be supplied with

- an object defining its own system message type, used to instantiate the in-tray, secretary and stub modules
- objects for the message number counter, in-tray, and secretary, which are just instantiations of generic modules
- a constant specifying its own address
- stub modules to replace the lower level modules
- *can_handle* and *handle* functions for the (skilled) secretary module to use

Only the last two of these require more than a very few lines of specification, and are easy to write and to check because they follow a very regular pattern.

This is a very small change to the specification. For the taxation system, for example, only three modules of the 22 needed any change, and the changes only affected a very small part of these three.

4.2 Semantics

The construction of the distributed system is comparatively simple. But its semantics seem much more of a problem. In "opening up" the hierarchies and apparently allowing arbitrary asynchronous communication we are immediately faced with the notorious

problem of interference which prevents us making reliable conclusions about concurrent systems based on the properties of their components.

But in fact our distributed system retains some important structural properties. Although the message system is capable of allowing communication between arbitrary nodes, it is only used in very particular ways:

- The communication within, say, the tax system is still hierarchical. That is, we did not introduce any new communication paths between tax offices. The only possible communications between tax offices are those between an office and its immediate superior or inferiors in the hierarchy.
- We only introduce "horizontal" communication paths for particular purposes, such as allowing a treasury office to report a tax payment to a tax office. If we can keep the number of these paths low we can deal with them individually.
- We have adopted the protocol that all messages are replied to. This means that we can rely on either obtaining an answer to a query or deciding, after some suitable wait, that it is "null".

We consider a number of requirements and see how we can validate the distributed system against them:

1. Taxes for taxpayers are calculated according to the current regime on the basis of their bases and roll information.
2. Reports collected by an office will correctly reflect the current information from its group of taxpayers and/or its subsidiary offices
3. Tax paid by a taxpayer at a treasury office will be correctly credited at the corresponding tax office

The first requirement is mostly about the calculations that are carried out within the "Group" specification. The group is still part of a provincial or district tax office and is unaffected by the distribution. If this property was true in the original, applicative specification it will, appropriately transformed, still be true.

There are some additional questions about the "current" regime, the identification of the taxpayer's roll and base information, but it is comparatively easy to check that there is a function to transmit a regime down the tax hierarchy (and to check its receipt) and that the one transmitted is the one used until a new one is received. Similarly we can check that roll and base information is properly installed.

The second requirement is another example of a function within the tax system. In the original, applicative specification it was stated explicitly. Hence we know its concurrent counterpart holds in the hierarchical specification. The problem comes now from two sources: interference from other activities, either within the tax system (e.g. a taxpayer declares their profits and changes their base) or from another system (e.g. the treasury reports payment of some tax by a taxpayer).

Information systems like this are typically not meant to deal in very precise ways with this kind of problem. It is not in general required, say, that an office "lock" its database against all other accesses while a report is compiled, only that its compilation does not affect those other activities. It is accepted that figures may vary according to other events occurring, by chance, just before or just after the report is compiled. The problem, in

this case, is one of finding a specification of the "current information" that is sufficiently loose!

A possible approach to this problem is to use partial orders on events as indicating causality [9], and to specify that the event of asking for a report from each district precedes the arrival of the request there, which precedes the report being sent back, which in turn precedes its receipt and merging with others. The "current information" is that existing at some point between the request arriving and the report's dispatch, and hence between the superior office's request and merge. But we can check informally that the requirement is met if we check that each superior office sends the appropriate message to each of its subordinate ones, that such messages are properly delivered, that the subordinate offices "handle" and correctly reply with the information requested, and that the responses are correctly merged. Much of this (the production of the report in the district office, the merging of reports at the province office) is already specified in the hierarchical system and reused (unchanged) in the distributed system.

The third requirement is an example of communication between systems, so it is a requirement that the separate hierarchical systems could not have been specified to meet. But we can decompose it into

1. Payment of tax at a treasury office will be correctly reported to the appropriate tax office.
2. (Report of) payment of tax to a tax office will be correctly credited

The second of these is already a property of the "Group" and is unchanged by the distribution. So we need to check:

- Each treasury office can receive tax payments.
- The correct information (amount, tax category, taxpayer) is sent to the correct tax office.
- The message system delivers messages to the correct recipient.
- Such messages can be correctly "handled" by tax offices, with the appropriate group functions called, and the appropriate acknowledgement sent.

(These need to be extended to allow for possible non-communication with checks for non-acknowledgement, resending of information, and recognition of duplicate inputs.)

We conclude that it is not feasible to have a general theory of such a distribution, but that if we have sufficient restrictions on possible communication paths, and suitable message protocols, then we can argue informally that requirements are met provided particular properties of the extra components we added for distribution are true. These are properties like

- The message system sends messages to their addressees
- Messages are correctly encoded/decoded
- Received messages are correctly "handled", i.e. the appropriate existing function is called with the correct parameters
- Messages are replied to with the correct response, or acknowledged

The point to notice about these properties are that they are easily stated requirements of, for example, the message system or the encode/decode functions, and hence can be part

of their specification, or they are easily checked by looking at a few lines of specification. The only possible exception to this is the "there is always a reply or acknowledgement" since its generation may conceivably be some way textually from the handling of the message it should be a response to, but in practice one can structure the specification so that it is clear. Hence these properties can be checked "by inspection" rather than by proof; they are amenable to quality control.

Or, of course, one can take a particularly critical property like "all messages are replied to or acknowledged" and perform the proof. In general we want to restrict proofs at this stage to critical properties, because of the problems of proof about concurrent systems that we remarked on earlier, and do as much as possible by quality control.

5 Conclusions

We achieved a number of aims. We were able to separate clearly the functional aspects of particular parts of the system (like tax accounting and report merging) from more organisational aspects. We were able to do this partly through adopting a "bottom-up" approach that allowed us to tackle one problem at a time. This also had a pedagogic purpose — it is particularly hard to do things top-down with people with little experience in formal specification. Much of what is done top-down in making things abstract and generic, and in dealing with many modules, is hard to motivate to such people, and makes their initial specifications hard to conceptualise. But as long as one has the confidence that things can be put together later the bottom-up approach has much to recommend it in keeping things simple for as long as possible, and in allowing separate parts to be worked on independently. In general, even with experienced people, it is often a good idea to look first at new, difficult problems regardless of where they will eventually appear in the specification structure.

We stated at the start of this paper that engineering involved composing entities with known properties in combinations with accompanying rules that allowed the properties of the combinations to be computed. This we have done for the applicative to imperative, sequential to concurrent transformations. We know exactly how to relate the properties of the result to the properties of the starting point. The method involves working initially with applicative specifications, perhaps refining these to more concrete versions, and perhaps even proving these refinements, or at least some important properties. Then there is a transformation step which is simple enough to be amenable to quality control. It might be automated, though in practice there are various options that can be chosen as to how exactly to structure the variables or channels being introduced.

The further step introduced in this paper from separate hierarchical systems to a combined and distributed system uses a number of standard generic modules. The changes to the existing system are very small and, again, open to quality control.

The introduction of asynchrony, and the opening of hierarchies for independent communications between their components, makes the system semantics more complicated, and much more difficult to relate to those of the synchronous, hierarchical system. But it seems that, given sufficient architectural constraints on the possible communication paths, and reasonable protocols that enable non-reception of messages to be decided, we can still relate asynchronous properties to synchronous ones in a reasonable manner, and

the required checks are again amenable to quality control. There is more work to be done to generalise and properly formalise these ideas.

The attempt to separate concerns and put things together later is not always so successful. We also looked, for example, at how the security aspects of the tax system could be specified, starting with the current policy. This uses a quite conventional system of groups of users with levels of functional access to the user-level functions and access rights for various functions and parts of the taxation database in each office.

We hoped at one point to be able to put a "shell" around the system so that user access could be controlled at the outer level without changing the specification within it. But this proved very difficult, mainly because at the abstract level it is difficult to say exactly what data is being accessed. It seemed inevitable that we would need to pass the user identities associated with top level transactions down to the level of the database accesses in order to validate them there. This is probably good practice anyway, as there is less of the system to validate against security leaks. But it involves a simple but rather tedious addition of an extra parameter to many functions.

References

1. The RAISE Method Group. *The RAISE Development Method.* BCS Practitioner Series. Prentice Hall, 1995.
2. Do Tien Dung, Le Linh Chi, Nguyen Le Thu, Phung Phuong Nam, Tran Mai Lien, and Chris George. Developing a Financial Information System. Technical Report 81, UNU/IIST, P.O.Box 3058, Macau, September 1996.
3. Do Tien Dung, Chris George, Hoang Xuan Huan, and Phung Phuong Nam. A Financial Information System. Technical Report 115, UNU/IIST, P.O.Box 3058, Macau, July 1997.
4. The RAISE Language Group. *The RAISE Specification Language.* BCS Practitioner Series. Prentice Hall, 1992.
5. J.V. Guttag. Abstract data types and the development of data structures. *CACM*, 20(6), June 1977.
6. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.
7. R. Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.
8. R.E. Milne. Transforming Axioms for Data Types into Sequential Programs. In *Proceedings of 4th Refinement Workshop.* Springer-Verlag, 1991.
9. Sigurd Meldal and David C. Luckham. Defining a Security Reference Architecture. Technical Report CSL-TR-97-728, Stanford University, June 1997.

Software Engineering Issues for Network Computing

Carlo Ghezzi and Giovanni Vigna

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci, 32
20133 Milano, Italy
ghezzi**Fehler! Textmarke nicht definiert.**

Abstract

The Internet is becoming the infrastructure upon which an increasing number of new applications is being developed. These new applications allow new services to be provided and even new business areas to be opened. The growth of Internet-based applications has been one of the most striking technological achievements of the past few years. In this paper we discuss some risks inherent in this growth. Rapid development and reduced time to market has probably been the highest priority concern for application developers. Use of unstable technology is also typical of such developments. So far, applications development was less concerned with the quality of the resulting products, such as reliability or modifiability. And developments seem to proceed without following a disciplined approach. We argue that these systems will become the legacy systems of the near future, when people will discover that their quality needs to be improved but, at the same time, modifications will be hard to make in an economical and reliable way. In this paper we discuss the needs for a software engineering approach to the development of network applications. In particular, we discuss a possible research agenda for software engineering research by looking at two specific areas: the Web and applications based on mobile code.

Keywords and phrases: Internet, World Wide Web, mobile computing, distributed systems, software engineering, software quality, software development process.

2. Introduction

Since the beginning of the 1990's, use of the Internet and the World Wide Web (WWW) have exploded, fostered by cheaper and higher performance hardware and communication media. Today, there are more than 50 million users from about 200 countries; the yearly growth rate has been more than 50%, and the number of forecasted users by the year 2000 is about 380 million [Kambil, Doing Business in the Wired World, IEEE Computer 30, 5, pp. 56-61, May 1977.].

In recent years, there has been a shift in the way the Internet is being used and in the way its potential is perceived both by technology developers and by users. It is not seen merely as the communication infrastructure that allows people to communicate in a fast, cheap, and reliable way. It is increasingly seen as the infrastructure upon which new services, new applications, and even new and previously unforeseen types of social processes are becoming possible. For example, electronic commerce will probably support new kinds of business and will change the way business is done. As another example, interactive distance learning and tutoring will probably change the way knowledge is transferred and will support new kinds of learning processes.

A new field is therefore emerging: network computing. By this, we mean computing where the computational infrastructure is a large set of autonomous and geographically distributed computers connected by the Internet. Although such infrastructure is a distributed system, and therefore the methods and techniques developed so far by the research community dealing with distributed computing can be viewed as foundational background for network computing, many new issues arise that make network computing different from distributed computing. This issue is discussed further in Section 2.

In this paper we discuss the current stage of network computing in a critical way, in order to understand the risks that are currently inherent in his growth. Rapid development and reduced time to market seem to be the major concerns that drive the developments of network applications. Furthermore, such applications are developed using either unstable or inappropriate technology. Applications are developed in an ad-hoc manner, without following disciplined design approaches, and often with little concern on their qualities, such as reliability or modifiability. We argue that these systems are likely to become the legacy systems of the near future, when people will discover that they will be difficult to maintain in an economical and reliable way. We see a similarity between the current situation and the one that existed in the sixties (see [E.W. Dijkstra, GOTO Statement Considered Harmful] and much of the work that was then spurred on by this paper), when the risks due to the lack of appropriate mathematical foundations, methods, and tools were recognized, and a suitable research agenda was set for software engineering to tame those risks.

In this paper, we dig into two specific important areas of the network computing domain: the Web and applications based on mobile code. For these two areas we discuss where the main risks are and outline a possible research agenda. This is not meant to be an exhaustive account of what is needed, but rather reflects a subjective viewpoint that is based on our work and some initial results that have been achieved by our research group.

3. Network computing

Network computing was defined as computing where the computational infrastructure is a network of geographically distributed computers connected by a telecommunication network. Such a network is composed of heterogeneous technologies: from those used to connect computers in a LAN, to those used to interconnect LANs geographically, to those used to connect mobile computing devices. The different technologies used to provide connectivity, spanning from fiber optics to different kinds of wireless connections, provide different levels of quality of services, e.g., in terms of performance and reliability.

Network interconnection is becoming a commodity. It is available anywhere and anytime, independent of the user's physical location. Wireless technology allows computational nodes to move, still being connected to the net. This supports mobile users, who may be using laptops or personal digital assistants (PDAs). As we will discuss in Section 4, the network infrastructure allows not only computing devices to move, but also software components to migrate over the network. Such components can be viewed as „software agents", which can have some levels of autonomy in achieving their specific goals, including the ability to move to different computing nodes.

The potential of this pervasive and ubiquitous infrastructure is enormous, and it is quite difficult to anticipate the way it will evolve and how far it will go. New applications and new services are announced almost every day. Although in many cases they promise more than they actually deliver, the speed and complexity of the evolution are indeed so high that they are difficult to dominate. It is therefore quite important to build a coherent framework of principles, abstractions, methods, and tools that would allow network computing to be understood and practiced in a systematic fashion [A. Fuggetta, G.P. Picco, and G. Vigna, *Understanding Code Mobility,* Politecnico di Milano Technical Report, June 1997, *Accepted for publication on IEEE Transactions on Software Engineering.*]. We claim that these are the challenges that software engineering must face in this context.

At the foundational level, we need to identify the theories that are needed to describe, reason about, analyze network computations, where the topology and structure of the computing layer can change dynamically, users can move, and computations can move. We need to identify security models to protect the sites from possible attacks of incoming agents; and, conversely, to protect agents from malicious sites. From a methodology viewpoint, we need to identify process models that are suitable to describing and managing applications developments for the new computing infrastructure, where applications grow in a largely independent way, no precise pre-planning is possible, and evolution/reconfiguration are the norm.

How can we deal with inherently chaotic, partly self-regulating systems? From the technology viewpoint, what languages can be provided to program applications? What tools can support interoperability? What tools can be defined to support service deployment, by allowing changes of evolving applications to be easily distributed to their users?

This wide spectrum of problems provides a real challenge for software engineering research. A number of efforts are already in place, but much more focused work is needed. The efforts we describe in this paper are only a small sample of what could be done in this area.

4. Software Engineering for WWW applications

From its first introduction in 1990 [T. Berners-Lee, R. Cailliau, A. Luotonen, H. Frystyk Nielsen and A. Secret, *The World Wide Web,* Communications of the ACM, vol. 37, num. 8, August 1994.], the World Wide Web (WWW) is evolving at a fast pace. The number of WWW sites is increasing as Internet users realize the benefits that stem from a globally interconnected hypermedia system. Each site, in fact, provides structured information as an intertwined net of hypertext resources; links can point both to local resources and to nonlocal resources belonging to other Web sites, thus providing a way to navigate from local to geographically distributed information sources. Companies, organizations, and academic institutions exploit the WWW infrastructure to provide customers and users with information and services. The expectations of both providers and consumers are driving R&D efforts aimed at improving the WWW technology. Examples are represented by the introduction of active contents in static hypertext pages by means of languages like Java [J. Gosling and H. McGilton, *The Java Language Environment: a White Paper,* Technical Report, Sun Microsystems, October 1995.] and JavaScript [D. Flanagan, *JavaScript – The Definitive Guide,* 2nd Edition, O'Reilly & Ass., January 1997.] and by the use of the Servlet technology [Sun Microsystems, *The Java Servlet API White Paper,* 1997.

22. Taligent Inc.,Building *Object-Oriented Frameworks,* A Taligent White Paper, 1994.

23. G. Vigna, *Paradigms and Technologies for Distributed Applications Development Based on Mobile Code,* PhD Thesis, Politecnico di Milano, 1998.

] to customize the behavior of Web servers. This technological evolution has promoted a shift in the intended use of the WWW. The Web infrastructure is going beyond the mere distribution of information and services; it is becoming a platform for generic, distributed applications in a worldwide setting.

This promising scenario is endangered by the lack of quality of most existing WWW-based applications. Although there is no well-defined and widely accepted notion of Web quality (and indeed, this would be a valuable research objective in its own), our claim is based on the following common observations that can be made as WWW users:

1. we know that a required piece of information is there in a certain WWW site, but we keep navigating through a number of pages without finding it;
2. we get lost in our navigation, i.e., we do not understand where we are in our search;
3. we keep encountering broken links;
4. the data we find are outdated (for example, we find the announcement of a „future" event that has already occurred);
5. duplicated information is inconsistent (for example, in a university Web providing pages in two language versions, say English and Italian, the same instructor has different office hours);
6. the navigation style is not uniform (for example, the „next page in the list" link is in the bottom right corner for some pages, and in the top left corner for others).

This is only a sample list. Items 3 to 5 of the list can be defined as Web flaws; they affect „correctness" of the Web. The others are more related to style issues, and affect usability. Furthermore, even if we start from a Web that does not exhibit these weaknesses, these are likely to occur as soon as the Web undergoes modifications. Thus maintenance of legacy Webs becomes more and more difficult, and Web quality becomes lower and lower. If we try to understand what the causes of these inconveniences are, we realize that they all boil down to the lack of application of systematic design principles and the use of inadequate (low-level) tools during development.

Most current WWW site designs are not guided by systematic design methodologies and do not follow well-defined development processes. Rather, they proceed in a very unstructured, ad-hoc manner. Developers focus very early, and predominantly, on low-level mechanisms that enable, for example, particular visual effects, without focusing on who the expected users are, what the conceptual contents of the information is, and how should the information be structured. In particular, they rarely focus on the underlying conceptual model of the information that should be made available through the Web. The lack of such conceptual model becomes evident to the users, who find it difficult to search the Web to retrieve the data they are interested in. In addition, even if design starts at a high level, from a conceptual model of the information to be made available, no design guidance nor adequate abstractions are available to help Web designers move down systematically towards an implementation, possibly being supported by suitable tools.

This situation reminds the childhood of software development when applications were developed without methodological support, without the right tools, simply on the basis of good common sense and individual skills. WWW site development suffers from a similar problem. Most WWW developers delve directly into the implementation phase, paying little or no attention to such aspects as requirements acquisition, specification, and design. Too often, implementation is performed by using a low-level technology, such as the Hypertext Markup Language (HTML) [Sun Microsystems, *The Java Servlet API White Paper,* 1997.
22. Taligent Inc.,Building *Object-Oriented Frameworks,* A Taligent White Paper, 1994.

23. G. Vigna, *Paradigms and Technologies for Distributed Applications Development Based on Mobile Code,* PhD Thesis, Politecnico di Milano, 1998.

]. Using the analogy with conventional software development, this approach corresponds to implementing applications through direct mapping of very informal designs (if any) into an assembly-level language. Furthermore, the lack of suitable abstractions makes it difficult to reuse previously developed artifacts, or to develop frameworks that capture the common structure of classes of applications and allow fast development by customization. Finally, the management of the resulting Web site is difficult and error prone, because change tracking and structural evolution must be performed directly at the implementation level. This problem is particularly critical since WWW systems, by their very nature, are subject to frequent updates and even redesigns.

Software research has provided methods for requirements acquisition, languages and methods for specification, design paradigms, technologies (such as object-oriented programming languages), and tools (e.g., integrated development environments) that provide systematic support to the software development process. In principle, their availability should allow software developers to deliver quality products in a timely and cost-effective manner. A similar approach has to be followed in order to bring WWW development out of its immaturity. The next two subsections discuss a possible approach to these problems.

3.1 A WWW Software process

The benefits of a well-defined and supported software process are well known [C. Ghezzi, M Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering,* Prentice Hall, 1991.]. As for conventional software, the development of a Web site should be decomposed into a number of phases: requirements analysis and specification, design, implementation. After the site has been implemented and delivered, its structure and contents are maintained and

evolved. By identifying these phases of the development process we do not imply any specific development process structure. Different process models (waterfall, spiral, prototype based) can be accommodated in the framework. Actually, the continuous and rapid changes in business, which will be reflected by the evolution of the corresponding WWW sites, is likely to favor flexible process lifecycles, based on rapid prototyping and continuous refinement. In the sequel, we briefly and informally outline the possible objectives of the different phases of WWW development, based on our own experience.

Requirements analysis and specification

During requirements analysis, the developer collects the needs of the stakeholders, in terms of *contents*, *structuring*, *access*, and *layout*. Contents requirements define the domain-specific information that must be made available through the Web site. Structuring requirements specify how contents must be organized. This includes the definition of *relationships* and *views*. Relationships highlight semantic connections among contents. For example, relationships could model generalization (*is-a*), composition (*is-composed-of*), or domain-dependent relationships. Views are perspectives on information structures that „customize" contents and relationships according to different use situations. Different views of the same contents could be provided to different classes of user (e.g., an abstract of a document can be made accessible to „external" users, while the complete document can be made accessible to „internal" users). Access requirements define the style of information access that must be provided by the Web site. This includes priorities on information presentation, indexing of contents, query facilities, and support for guided tours over sets of related information. Layout requirements define the general appearance properties of the Web site, such as emphasis on graphic effects vs. text-based layout.
We argue that existing tools supporting requirements specification and traceability of requirements through all development artifacts can be used in this context too. Further research is needed both to extend the above framework and to identify the additional specific features that a tool supporting requirements for Web based applications should exhibit.

Design

Based on the requirements, the design phase defines the overall structure of a WWW site, describing how information can be organized and how users can navigate across it. A careful design activity should highlight the fundamental constituents of a site; it should abstract away from low-level implementation details, and should allow the designer to identify recurring structures and navigation patterns to be reused [F. Garzotto, L. Mainetti, and P. Paolini, *Information Reuse in Hypermedia Applications,* Proceedings of ACM Hypertext '96, Washington DC, ACM Press, March 1996.]. As such, a good design can survive the frequent changes in the implementation, fostered by –say-- the appearance of new technologies.

Being largely implementation-independent, the design activity can be carried out using notations and methodologies that are not primarily Web-oriented. Any design methodology for hypermedia applications could be used; e.g., HDM [F. Garzotto, L. Mainetti, and P. Paolini, *Hypermedia Design, Analysis, and Evaluation Issues,* Communications of the ACM, Vol. 38, No. 8, August 1995.], RMDM [V. Balasubramanian, T. Isakowitz, and E. A. Stohr, *RMM: A Methodology for Structured Hypermedia Design,* Communications of the ACM, 38(8), August 1995.], or OOHDM [Schwabe and G. Rossi, *From Domain Models to Hypermedia Applications: An Object-Oriented Approach,* Proceedings of the International Workshop on Methodologies for Designing and]. Our experience is based on the adoption of

the HDM (Hypertext Design Model) notation [F. Garzotto, L. Mainetti, and P. Paolini, *Hypermedia Design, Analysis, and Evaluation Issues,* Communications of the ACM, Vol. 38, No. 8, August 1995.].

In designing a hypermedia application, HDM distinguishes between the *hyperbase layer* and the *access layer*. The hyperbase layer is the backbone of the application and models the information structures that represent the domain, while the access layer provides entry points to access the hyperbase constituents.
The hyperbase consists of *entities* connected by *application links*. Entities are structured pieces of information. They are used to represent conceptual or physical objects of the application domain. An example of entity in a literature application is „Writer". *Application links* are used to describe domain-specific, non-structural relationships among different entities (e.g., an application link from a „writer" to the „novels" he wrote). Entities are structured into *components*, i.e., clusters of information that are perceived by the user as conceptual units (for example, a writer's „biography"). Complex components can be structured recursively in terms of other components. Information contained in components is modelled by means of *nodes*. Usually, components contain just one node, but more than one node can be used to give different or alternative views (*perspectives*, in HDM) of the component information (e.g., to describe a piece of contents in different languages, or to present it in a „short" vs. an „extended" version). Navigation paths inside an entity are defined by means of *structural links*, which represent structural relationships among *components*. Structural links may, for example, define a tree structure that allows the user to move from a root component (for example, the data-sheet for a novel) to any other component of the same entity (e.g., credits, summary, reviews, etc.)

Once entities and components are specified, as well as their internal and external relationships, the access layer defines a set of *collections* that provide users with the structures to access the hyperbase. A collection groups a number of „members", in order to make them accessible. Members can be either hyperbase elements or other collections (nested collections). Each collection owns a special component called *collection center* that represents the starting point of the collection. Examples of collections are *guided tours*, which support linear navigation across members (through next/previous, first/last links), or *indexes*, where the navigation pattern is from the center to the members and viceversa. For example, a guided tour can be defined to navigate across all horror novels, another one can represent a survey of 14^{th} century European writers.

Implementation

The implementation phase creates an actual Web site from the site design. As a first step, the elements and relationships highlighted during design are mapped onto the constructs provided by the chosen implementation technology. As a second step, the site is *populated*. The actual information is inserted by instantiating the structures defined in the previous step and the cross-references representing structural and application links among the elements. Collections are then created to provide structured access to the hyperbase contents. The third step is *delivery*. The site implementation must be made accessible using standard WWW technologies, namely Web browser like Netscape's Navigator or Microsoft's Internet Explorer that interact with servers using the Hypertext Transfer Protocol (HTTP). This can be achieved by translating the site implementation into a set of files and directories that are served by a number of „standard" WWW servers (also called *http daemons* in the UNIX jargon).

The standard tools available today to implement the Web are rather low-level and semantically poor. The basic abstractions available to Web developers are:

- *HTML pages*, i.e., text files formatted using a low-level markup language;
- *directories*, i.e., containers of pages; and
- *references*, i.e., strings of text embedded in HTML tags that denote a resource (e.g., an HTML page) using a common naming scheme.

There are no systematic methods nor linguistic constructs to define the mapping of the types that define the semantics of the application domain (entities) onto implementation-level types (pages). There are no constructs to define complex information structures, like sets of pages with particular navigational patterns, such as lists of pages or indexes. Such structured sets of information must be realized manually by composing the existing constructs and primitives. In addition, there is no way to create document templates and mechanisms to extend existing structures by customization. The development of a set of documents exhibiting the same structure is carried out in an *ad hoc* manner by customizing manually sample prototypes. There are no constructs or mechanisms to specify different views of the same information and to present such views depending on the access context. This hampers effective reuse of information. The only form of reuse is *by copy*. Some authoring tools like Microsoft's FrontPage [Microsoft Corp., *FrontPage Home Page,* Fehler! **Textmarke nicht definiert.**] and NetObject's Fusion [NetObjects Inc., *Fusion Home Page,* Fehler! **Textmarke nicht definiert.**] try to overcome some of these limitations by providing a site-level view on the information hyperbase. Nonetheless, these tools are strictly based on the low-level concepts of HTML pages and directories. As a consequence, the developer is faced with a gap between the high level concepts defined during design and the low-level constructs available for implementation.

The situation gets worse in the maintenance phase. Web sites have an inherently dynamic nature. Contents and their corresponding structural organization may be changed continuously. Therefore, maintenance is a crucial phase, even more than in the case of conventional software applications. As for conventional software, we can classify maintenance into three categories: *corrective, adaptive*, and *perfective maintenance* [C. Ghezzi, M Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, Prentice Hall, 1991.]. Corrective maintenance is the process of correcting errors that exist in the Web site implementation. Examples are represented by internal dangling references, errors in the indexing of resources, or access to outdated information (as in the case of published data with an expiration date). Adaptive maintenance involves adjusting the Web site to changes in the outside environment. A notable example is represented by verification of the references to documents and resources located at different sites. Outbound links become dangling as a consequence of events over which the site developer has no control. Thus, maintenance is a continuous process. Perfective maintenance involves changing the Web site in order to improve the way contents are structured or presented to the end user. Changes may be fostered by the introduction of new information or the availability of new technologies. Perfective maintenance should reflect updates to the requirements and design documents. Maintenance in general, and perfective maintenance in particular, is by far the activity that takes most of the development effort.

Presently, Web site maintenance is carried out using tools like link verifiers or syntax checkers that operate directly on the low-level Web site implementation. This approach may be suitable for some cases of corrective and adaptive maintenance, but does not provide effective support for tasks that involve knowledge of the high-level structure of the Web site. For example, since reuse is achieved by copy, modifying a reused component, like a recurring

introduction paragraph for a number of documents, involves the identification of every use of the component and its consistent update. In a similar way, modification of the structure or style of a set of similar documents requires updates in all instances. For example, if we decide that the background color of the summary page of all „horror" novels must be changed to purple, this requires consistent change of all files representing such summaries. More generally, since perfective maintenance may require a modification of the structure and organization of information, it should be supported by a structural view of the site and of the relationships between design elements and their implementation constructs. These relationships are of paramount importance because they allow the developer to reflect design changes onto the implementation and viceversa. Standard Web technologies do not provide any means to represent these relationships and the high level organization of information. Another problem concerns maintenance of hypertext references. In the standard WWW technology, references are just strings embedded inside the HTML code of pages; they do not have the status of first-class objects. Therefore, their management and update is an error prone activity.

3.2 The WOOM approach

An example of what could be done to support Web design is given by a project we are currently carrying out in our group. In this project, we developed a WWW object-oriented modeling framework, called WOOM – *Web Object Oriented Model*. WOOM provides concepts, abstractions, and tools that help in the mapping from high-level design of a Web site (e.g., in HDM) into an implementation that uses „standard" WWW technology.

In WOOM, a site is first modeled by introducing a number of entities and relationships, using an object-oriented notation. Entities can be organized in an inheritance hierarchy. In addition, WOOM predefines the basic implementation types that can be used to implement WWW sites. This includes hyperpages, containers (which can contain hyperpages), and elements (the constituents of a page)[1]. An entity implementation is described by inheriting from the entity and from an implementation type (e.g., a page). A WWW instance is described as a DAG whose node types are implementation types (or subtypes thereof). The DAG allows an information item (e.g., a page or a container) to be shared by different contexts. For example, one can describe that the page representing a novelist (say, Patricia Highsmith) belongs both to the container which groups the American contemporary writers and the container grouping crime-story writers. The process of publication of the DAG, which generates the target Web site in HTML, takes care of generating two different instances of Patricia Highsmith's page. In the context of American contemporary writers, its „next writer button" might, say, refer to Saul Bellow' page. In the context of crime-story writers, such a button might refer to, say, George Simenon's page. Since this done automatically by the publication tool, consistency is automatically preserved (only one instance of the writer's data is kept), and maintenance is greatly facilitated. In addition, to improve currency of the published data, the prototype version of WOOM that is being developed allows such data to be kept in a database, and the values to be extracted from it at publication time. Moreover, the publication process is quite sophisticated. Basically, it consists of a recursive traversal of the DAG. In such traversal it propagates attributes that allow for powerful customization of the information to be published in the different contexts, based on the attributes of the object encountered during traversal. For

[1] This is a simplified and incomplete list of the implementation types provided by WOOM. Basically, these types define the components that can be defined using HTML.

example, we might wish to specify that the background color of horror story pages should be yellow, while the background of American contemporary writers should depict the American flag. This can be specified by two different actions setting the background, performed by the publishing algorithm as the relevant pages are encountered during the recursive traversal. As another example, one attribute might be the expiration date of a page. During traversal, the page with an expired validity date would not be published in target web page. In general, it is possible to define events, whose occurrence would trigger the publication process to be performed as changes occurred which would require modification of the target site.

An important result of this approach is that it clearly separates the description of the data from the way the data are presented through the Web interface. The same data can be presented differently in different contexts. This separation not only helps in designing the application, but also provides support to changes and Web site evolution.

5. Mobile computing

As we mentioned, a new class of network applications is emerging, which assumes that users can move and software components can also migrate over the net. This area of computing is often informally denoted as „mobile computing".

The idea that software can migrate is not new. In particular, it has been exploited by several distributed systems in order to support load balancing. Mobile computing, however, differs from distributed computing in many respects [A. Fuggetta, G.P. Picco, and G. Vigna, *Understanding Code Mobility,* Politecnico di Milano Technical Report, June 1997, *Accepted for publication on IEEE Transactions on Software Engineering.*]. First, traditional distributed computing systems deal with a set of machines connected by a local network, whereas in mobile computing mobility is exploited at a much larger scale (the Internet scale); hosts are heterogeneous, they are managed by different authorities, they are connected by heterogeneous links. Secondly, mobility is seldom supported in distributed systems. In the particular cases where it is supported, it is not provided as a feature given to the programmer to be exploited in achieving particular tasks. Rather, it is used to allow components to be automatically relocated by the system to achieve load balancing. This, however, is not visible to the applications' programmer, since a software layer is often provided on top of the network operating system to hide the concept of physical locality of software components. On the other hand, in mobile computing programming is location aware and mobility is under the programmer's control. Components can be moved to achieve specific goals, such as accessing specific resources.

There are many expectations from mobile computing; we fear, however, that they are a bit premature, and probably unjustified by the current level of maturity of the field. The technology to support mobility is still in its infancy, there are no methods to follow in designing applications based on mobility, and it still unclear which are the applications that can really benefit from mobility.

Concerning the available technology, let us consider programming languages supporting mobility. The concepts and terminology are confused, and often it is hard to compare precisely what different languages provide to support mobility. For example, some languages provide

facilities to simply move the code; others allow the code and (part of) the state to be moved[2]. The latter case is what one would expect to have to implement agents that move on the network to perform autonomous tasks. Unfortunately, however, the best known and most widely available example of network programming language (Java) does not support this kind of mobility, and therefore it is hard to implement network agents.

An important aspect of mobile languages is the way they support secure computations. Most current language proposals do not deal explicitly with this issue, which is often left as „future work". Instead, our viewpoint is that security should be one of the cornerstones of a language design. There are two facets of the security problem. First, it is necessary that the site which hosts execution of an incoming component protects itself from malicious attacks. This is traditionally recognized as a problem from researchers, and is rather well studied. Second, its is necessary to protect the incoming component from attacks from malicious hosts. For example, imagine the case where the component is an agent that migrates over the network to perform some critical business operation for a user, such as finding the best options for investment. A malicious site might change the data accumulated in the agent's state to make the options provided by the site look better than what was found by visiting other sites. A first attempt to deal with security issues in the design of a mobile language is provided in [G. Vigna, *Paradigms and Technologies for Distributed Applications Development Based on Mobile Code*, PhD Thesis, Politecnico di Milano, 1998.].

Concerning design of mobile applications, one would like to be able to be provided with a number of design paradigms among which to choose to structure an application. Here too traditional distributed systems and mobile computing systems differ from one another. In traditional distributed computing, applications are mainly designed by using the *client-server* paradigm, using linguistic facilities like remote procedure call. Code mobility supports a wider range of paradigms, among which we can consider the following sample:

- *remote evaluation*, where code is uploaded remotely to perform its task, and the results are sent back to the originator site;
- *code-on-demand*, where code is downloaded from a remote site;
- *autonomous agent*, where a software component moves along with its state.

In principle, it would be helpful to be able to analyze the tradeoffs among the different solutions based on different paradigms at the design level, before proceeding to an implementation. For example, [A. Carzaniga, G. P. Picco, and G. Vigna, Designing Distributed Applications with Mobile Code Paradigms, in Proceedings of the 19th International Conference on Software Engineering, Boston, 1997.] discusses a set of paradigms and evaluates their tradeoffs in an example of a distributed information retrieval application. The tradeoffs are evaluated in terms of a simple quantitative quality measure: network traffic. The case study shows that, in general, there is no definite winner among the different paradigms, but rather the choice depends on a number of parameters that characterize the specific problem instance.

These are just some initial steps in the direction of guiding design of mobile code applications by a systematic development process. More experiments are needed before we can identify a

[2] For a comprehensive survey and assessment of currently available mobile computing languages, the reader can refer to [G. Cugola, C. Ghezzi, G.P. Picco, G. Vigna, „Analyzing Mobile Code Languages", in *Mobile Object Systems: Towards the Programmable Internet*, Jan Vitek and Christian Tschudin, eds., Lecture Notes on Computer Science, Springer Verlag, April 1997.]

set of generally useful methods and techniques, and provide tools to support them. Besides distributed information retrieval, the areas in which mobility might be explored include:

- active documents, i.e., documents that include the capability of executing programs;
- advanced telecommunications services;
- remote configuration, control and maintenance of devices;
- support to distributed cooperation and coordination (e.g., distributed workflow);
- active networks, i.e., the ability of „programming" the network according to the application's needs.

6. Conclusions

Network computing is a rapidly evolving field, which is raising much interests, both in industry and in research institutions. It is a very promising field but, at this stage of maturity, it is perhaps raising too many unjustified expectations. Growth is both chaotic and exciting. We see many interesting things being done in practice which are not backed up by adequate methods and tools. A challenge exists for software engineering research to evaluate critically how things are done today in order to identify a possible comprehensive approach to the development of network applications. The „just do it" approach that seems to be behind the current efforts are simply inadequate to reach the desired levels of quality standards of the resulting applications, for example in terms of reliability and ease of change. We must, of course, keep into account what makes network applications different from most traditional applications. In particular, their intrinsic levels of flexibility, autonomy, decentralization, and continuous change that cannot be pre-planned centrally. These properties must eventually be combined with the necessary discipline that allows the desired level of reliability to be reached in a measurable and economical way.

References

1. V. Balasubramanian, T. Isakowitz, and E. A. Stohr, *RMM: A Methodology for Structured Hypermedia Design*, Communications of the ACM, 38(8), August 1995.
2. T. Berners-Lee, R. Cailliau, A. Luotonen, H. Frystyk Nielsen and A. Secret, *The World Wide Web*, Communications of the ACM, vol. 37, num. 8, August 1994.
3. A. Carzaniga, G. P. Picco, and G. Vigna, *Designing Distributed Applications with Mobile Code Paradigms*, in Proceedings of the 19th International Conference on Software Engineering, Boston, 1997.
4. F. Coda, C. Ghezzi, G. Vigna, F. Garzotto, Towards a Software Engineering Approach to Web Site Development, *9th International Workshop on Software Specification and Design* (IWSSW-9), Ise-shima, Japan, 1998.
5. G. Cugola, C. Ghezzi, G.P. Picco, G. Vigna, „Analyzing Mobile Code Languages", in *Mobile Object Systems: Towards the Programmable Internet*, Jan Vitek and Christian Tschudin, eds., Lecture Notes on Computer Science, Springer Verlag, April 1997.
6. E.W. Dijkstra, GOTO Statement Considered Harmful
7. D. Flanagan, *JavaScript – The Definitive Guide*, 2nd Edition, O'Reilly & Ass., January 1997.

8. A. Fuggetta, G.P. Picco, and G. Vigna, *Understanding Code Mobility*, Politecnico di Milano Technical Report, June 1997, *Accepted for publication on IEEE Transactions on Software Engineering*.

9. F. Garzotto, L. Mainetti, and P. Paolini, *Hypermedia Design, Analysis, and Evaluation Issues*, Communications of the ACM, Vol. 38, No. 8, August 1995.

10. F. Garzotto, L. Mainetti, and P. Paolini, *Information Reuse in Hypermedia Applications*, Proceedings of ACM Hypertext '96, Washington DC, ACM Press, March 1996.

11. F. Garzotto, P. Paolini, and D. Schwabe, *HDM – A Model-Based Approach to Hypertext Application Design*, ACM Transactions on Information System, Vol. 11, No. 1, January 1993.

12. C. Ghezzi and M. Jazayeri, *Programming Language Concepts*, 3rd edition, Wiley, 1997.

13. C. Ghezzi, M Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, Prentice Hall, 1991.

14. C. Ghezzi and G. Vigna, „Mobile Code Paradigms and Technologies: A Case Study", Proceedings of the *First International Workshop on Mobile Agents 97* (MA'97), Berlin, Germany, 7-8 April 1997.

15. J. Gosling and H. McGilton, *The Java Language Environment: a White Paper*, Technical Report, Sun Microsystems, October 1995.

16. Kambil, Doing Business in the Wired World, IEEE Computer 30, 5, pp. 56-61, May 1977.

17. Microsoft Corp., *FrontPage Home Page*, **Fehler! Textmarke nicht definiert.**

18. NetObjects Inc., *Fusion Home Page*, **Fehler! Textmarke nicht definiert.**

19. D. Ragget, *Hypertext Markup Language 3.2 Reference Specification*, W3C recommendation, 1996.

20. Schwabe and G. Rossi, *From Domain Models to Hypermedia Applications: An Object-Oriented Approach*, Proceedings of the International Workshop on Methodologies for Designing and Developing Hypermedia Applications, Edimburgh, September 1994.

21. Sun Microsystems, *The Java Servlet API White Paper*, 1997.

22. Taligent Inc.,*Building Object-Oriented Frameworks*, A Taligent White Paper, 1994.

23. G. Vigna, *Paradigms and Technologies for Distributed Applications Development Based on Mobile Code*, PhD Thesis, Politecnico di Milano, 1998.

A Two-Layered Approach to Support Systematic Software Development

Maritta Heisel[1] and Stefan Jähnichen[2,3]

[1] Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Institut für Verteilte Systeme
D-39016 Magdeburg, Germany
email: heisel@cs.uni-magdeburg.de
[2] FG Softwaretechnik
Technische Universität Berlin, Sekr. FR 5-6
Franklinstr. 28/29
D-10587 Berlin, Germany
jaehn@cs.tu-berlin.de
[3] GMD FIRST
Rudower Chaussee 5
12489 Berlin, Germany

Abstract. We present two concepts that help software engineers to perform different software development activities systematically. The concept of an *agenda* serves to represent technical process knowledge. An agenda consists of a list of steps to be performed when developing a software artifact. Each activity may have associated a schematic expression of the language in which the artifact is expressed and validation conditions that help detect errors. Agendas provide methodological support to their users, make development knowledge explicit and thus comprehensible, and they contribute to a standardization of software development activities and products.
The concept of a *strategy* is a formalization of agendas. Strategies model the development of a software artifact as a problem solving process. They form the basis for machine-supported development processes. They come with a generic system architecture that serves as a template for the implementation of support tools for strategy-based problem solving.

Keywords: Software engineering methodology, process modeling, formal methods

1 Introduction

Software engineering aims at producing software systems in a systematic and cost-effective way. Two different aspects are of importance here: first, the *process* that is followed when producing a piece of software, and second, the various intermediate *products* that are developed during that process, e.g., requirements documents, formal specifications, program code, or test cases.

To date, research on the process aspects of software engineering concentrates on the management of large software projects, whereas research on the product aspects of software engineering concentrates on developing appropriate languages to express the various software artifacts, e.g., object-oriented modeling languages, architectural description languages, specification or programming languages.

The work presented in this paper is intended to fill a gap in current software engineering technology: it introduces concepts to perform the *technical* parts of software processes in a systematic way. By ensuring that the developed products fulfill certain pre-defined quality criteria, our concepts also establish an explicit link between processes and products.

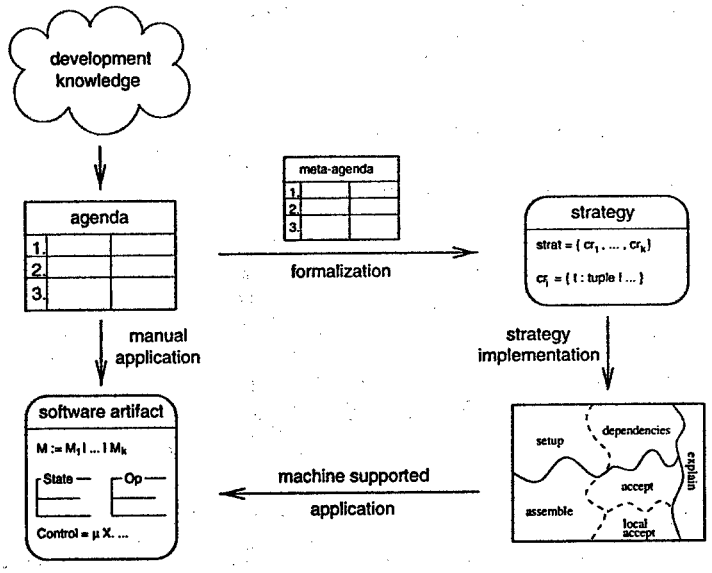


Fig. 1. Relation between agendas and strategies

We wish to systematically exploit existing software development knowledge, i.e., the problem-related fine-grained knowledge acquired by experienced software engineers that enables them to successfully produce the different software engineering artifacts. To date, such expert knowledge is rarely made explicit. As a consequence, it cannot be re-used to support software processes and cannot be employed to educate novices. Making development knowledge explicit, on the other hand, would

- support re-use of this knowledge,
- improve and speed up the education of novice software engineers,
- lead to better structured and more comprehensible software processes,
- make the developed artifacts more comprehensible for persons who have not developed them,
- allow for more powerful machine support of development processes.

Agendas and *strategies* help achieve these goals. An agenda gives guidance on how to perform a specific software development activity. It informally describes the different steps to be performed. Agendas can be used to structure quite different activities in different contexts.

Strategies are a formalization of agendas. They aim at machine supported development processes. The basic idea is to model software development tasks as problem solving processes. Strategies can be implemented and supplied with a generic architecture for systems supporting strategy-based problem solving.

Figure 1 shows the relation between agendas and strategies. First, the development knowledge used by experienced software engineers must be made explicit. Expressed as

an agenda, it can be employed to develop software artifacts independently of machine support. If specialized machine support is sought for, the agenda can be formalized as a strategy. Such a formalization can be performed systematically, following a meta-agenda. Implemented strategies provide machine support for the application of the formalized knowledge to generate software artifacts. In general, the steps of an agenda correspond to subproblems of a strategy.

Agendas and strategies are especially suitable to support the application of formal techniques in software engineering. Formal techniques have the advantage that one can positively guarantee that the product of a development step enjoys certain semantic properties. In this respect, formal techniques can lead to an improvement in software quality that cannot be achieved by traditional techniques alone.

In the following two sections, we present agendas and strategies in more detail. Related work is discussed in Section 4, and conclusions are drawn in Section 5.

2 Agendas

An agenda is a list of steps to be performed when carrying out some task in the context of software engineering. The result of the task will be a document expressed in a certain language. Agendas contain informal descriptions of the steps. With each step, schematic expressions of the language in which the result of the activity is expressed can be associated. The schematic expressions are instantiated when the step is performed. The steps listed in an agenda may depend on each other. Usually, they will have to be repeated to achieve the goal.

Agendas are not only a means to guide software development activities. They also support quality assurance because the steps of an agenda may have validation conditions associated with them. These validation conditions state necessary semantic conditions that the artifact must fulfill in order to serve its purpose properly. When formal techniques are applied, the validation conditions can be expressed and proven formally. Since the validation conditions that can be stated in an agenda are necessarily application independent, the developed artifact should be further validated with respect to application dependent needs.

2.1 An Agenda for Formally Specifying Safety-Critical Software

To illustrate the agenda concept, we present a concrete agenda that supports the formal specification of software for safety-critical applications. Because we want to give the readers a realistic impression of agendas, we present the agenda unabridged and give a brief explanation of the important aspects of software system safety and the language and methodology we use to specify safety-critical software.

The systems we consider in the following consist of a technical process that is controlled by dedicated system components being at least partially realized by software. Such a system consists of four parts: the *technical process*, the *control component*, *sensors* to communicate information about the current state of the technical process to the control component, and *actuators* that can be used by the control component to influence the behavior of the technical process.

Two aspects are important for the specification of software for safety-critical systems. First, it must be possible to specify behavior, i.e. how the system reacts to incoming

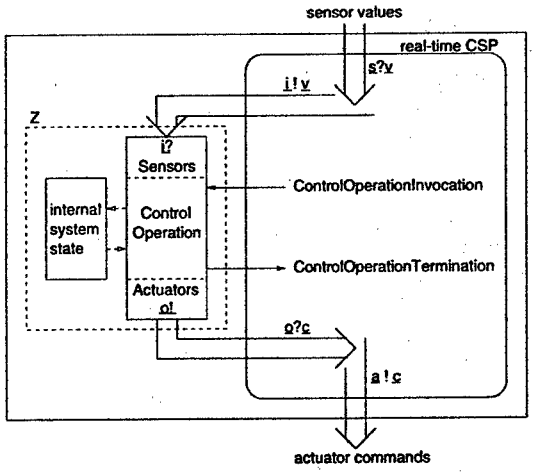


Fig. 2. Software Control Component for Passive Sensors Architecture

events. Second, the structure of the system's data state and the operations that change this state must be specified. We use a combination of the process algebra real-time CSP [Dav93] and the model-based specification language Z [Spi92] to specify these different aspects.

In [Hei97,HS96] we have described the following principles of the combination of both languages in detail: For each system operation *Op* specified in the Z part of a specification, the CSP part is able to refer to the events *OpInvocation* and *OpTermination*. For each input or output of a system operation defined in Z, there is a communication channel within the CSP part onto which an input value is written or an output value is read from. The dynamic behavior of a software component may depend on the current internal system state. To take this requirement into account, a process of the CSP part is able to refer to the current internal system state via predicates which are specified in the Z part by schemas.

There are several ways to design safety-critical systems, according to the manner in which activities of the control component take place, and the manner in which system components trigger these activities. These different approaches to the design of safety-critical systems are expressed as *reference architectures.*

We present an agenda for a reference architecture where all sensors are passive, i.e., they cannot trigger activities of the control component, and their measurements are permanently available. This architecture is often used for monitoring systems, i.e., for systems whose primary function is to guarantee safety. Examples are the control component of a steam boiler whose purpose it is to ensure that the water level in the steam boiler never leaves certain safety limits, or an inert gas release system, whose purpose is to detect and extinguish fire.

Figure 2 shows the structure of a software control component associated with the passive sensors architecture. Such a control component contains a single control operation,

which is specified in Z, and which is executed at equidistant points of time. The sensor values $\underline{v}$ coming from the environment are read by the CSP control process and passed on to the Z control operation as inputs. The Z control operation is then invoked by the CSP process, and after it has terminated, the CSP control process reads the outputs of the Z control operation, which form the commands $\underline{c}$ to the actuators. Finally, the CSP control process passes the commands on to the actuators.

Agendas are presented as tables with the following entries for each step:

- a numbering for easy reference,
- an informal description of the purpose of the step,
- a schematic expression that proposes how to express the result of the step in the language used to express the document,
- possibly some informal or formal validation conditions that help detect errors.

The agenda for the passive sensors architecture is presented in Tables 1 and 2, where informal validation conditions are marked "o", and formal validation conditions are marked "⊢". The dependencies between the steps are shown in Figure 3.

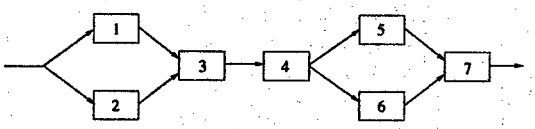


Fig. 3. Dependencies of steps of agenda for passive sensors architecture

The agenda gives instructions on how to proceed in the specification of a software-based control component according to the chosen reference architecture. Usually, different phases can be identified for processes expressed as an agenda. The first phase is characterized by the fact that high-level decisions have to be taken. For these decisions, no validation conditions can be stated. In our example, these are the Steps 1 and 2. In the second phase, the language templates that can be proposed are fairly general (for example, we cannot say much more than that schemas should be used to define the internal system states and the initial states), but it is possible to state a number of formal and informal validation conditions. In our example, the second phase consists of Steps 3 and 4.

In the third and last phase of an agenda, the parts of the document developed in the earlier phases are assembled. This can be done in a routine or even completely automatic way. Consequently, no validation conditions are necessary for this phase. In our example, the third phase consists of Steps 5 and 6. Step 7 allows specifiers to add specification text, if this is necessary for the particular application. The example shows that

- the agenda is fairly detailed and provides non-trivial methodological support,
- the structure of the specification need not be developed by the specifier but is determined by the agenda,
- the schematic expressions proposed are quite detailed,
- the validation conditions that help avoid common errors are tailored for the reference architecture and the structure of its corresponding specification.

No.	Step	Schematic Expressions	Validation Conditions
1	Model the sensor values and actuator commands as members of Z types.	$Type ::= \ldots$	
2	Decide on the operational modes of the system.	$MODE ::= Mode1 \mid \ldots \mid ModeK$	
3	Define the internal system states and the initial states.	*InternalSystemState* $mode : MODE$ $\ldots$ --- $\ldots$ *InternalSystemStateInit* $InternalSystemState'$ --- $\ldots$	o The internal system state must be an appropriate approximation of the state of the technical process. ⊢ The internal state must contain a variable corresponding to the operational mode. o Each legal state must be safe. ⊢ There must exist legal initial states. o The initial internal states must adequately reflect the initial external system states.
4	Specify an internal Z operation for each operational mode.	$Sensors \mathrel{\widehat=}$ $[InternalSystemState;$ $in1? : SType1; \ldots; inN? : STypeN \mid$ $\langle consistency / redundancy \rangle]$ $Actuators \mathrel{\widehat=}$ $[InternalSystemState';$ $out1! : AType1; \ldots; outM! : ATypeM \mid$ $\langle derivation\ of\ commands \rangle]$ $OpModeJ \mathrel{\widehat=} [\Delta InternalSystemState;$ $Sensors; Actuators \mid \ldots]$	⊢ The only precondition of the operation corresponding to a mode is that the system is in that mode. ⊢ For each operational mode and each combination of sensor values there must be exactly one successor mode. ⊢ Each operational mode must be reachable from an initial state. ⊢ There must be no redundant modes.

Table 1. Agenda for the passive sensors architecture, part 1

No.	Step	Schematic Expressions
5	Define the Z control operation.	*Control* $\Delta InternalSystemState$ *Sensors*; *Actuators* --- $mode = Mode1 \Rightarrow OpMode1$ $\wedge \ldots \wedge$ $mode = ModeK \Rightarrow OpModeK$
6	Specify the control process in real-time CSP.	$ControlComponent \mathrel{\widehat{=}} SystemInitExec \rightarrow ControlComp_{READY}$ $ControlComp_{READY} \mathrel{\widehat{=}} \mu X \bullet$ $((sensor1?valueS1 \rightarrow in1!valueS1 \rightarrow Skip \parallel \ldots \parallel$ $sensorN?valueSN \rightarrow inN!valueSN \rightarrow Skip);$ $ControlInvocation \rightarrow ControlTermination \rightarrow$ $(out1?valueA1 \rightarrow actuator1!valueA1 \rightarrow Skip \parallel \ldots \parallel$ $outM?valueAM \rightarrow actuatorM!valueAM \rightarrow Skip)$ $\parallel Wait\ INTERVAL); X$
7	Specify further requirements if necessary.	

Table 2. Agenda for the passive sensors architecture, part 2

2.2 Agenda-Based Development

In general, working with agendas proceeds as follows: first, the software engineer selects an appropriate agenda for the task at hand. Usually, several agendas will be available for the same development activity, which capture different approaches to perform the activity. This first step requires a deep understanding of the problem to be solved. Once the appropriate agenda is selected, the further procedure is fixed to a large extent. Each step of the agenda must be performed, in an order that respects the dependencies of steps. The informal description of the step informs the software engineer about the purpose of the step. The schematic language expressions associated with the step provide the software engineer with templates that can just be filled in or modified according to the needs of the application at hand. The result of each step is a concrete expression of the language that is used to express the artifact. If validation conditions are associated with a step, these should be checked immediately to avoid unnecessary dead ends in the development. When all steps of the agenda have been performed, a product has been developed that can be guaranteed to fulfill certain application-independent quality criteria.

Agenda-based development of software artifacts has a number of characteristics:

- **Agendas make software processes explicit, comprehensible, and assessable.** Giving concrete steps to perform an activity and defining the dependencies between the steps make processes explicit. The process becomes comprehensible for third parties because the purpose of the various steps is described informally in the agenda.
- **Agendas standardize processes and products of software development.** The development of an artifact following an agenda always proceeds in a way consistent with the steps of the agenda and their dependencies. Thus, processes supported by agendas are standardized. The same holds for the products: since applying an agenda results in instantiating the schematic expressions given in the agenda, all products developed with an agenda have the same structure.

- **Agendas support maintenance and evolution of the developed artifacts.**
  Understanding a document developed by another person is much less difficult when the document was developed following an agenda than without such information. Each part of the document can be traced back to a step in the agenda, which reveals its purpose. To change the document, the agenda can be "replayed". The agenda helps focus attention on the parts that actually are subject to change. In this way, changing documents is greatly simplified, and it can be expected that maintenance and evolution are less error-prone when agendas are used.
- **Agendas are a promising starting point for sophisticated machine support.**
  First, agendas can be formalized and implemented as strategies, see Section 3. But even if a formal representation of development knowledge is not desired, agendas can form the basis of a process-centered software engineering environment (PSEE) [GJ96]. Such a tool would lead its users through the process described by the agenda. It would determine the set of steps to be possibly performed next and could contain a specialized editor that offers the user the schematic language expressions contained in the agenda. The user would only have to fill in the undefined parts. Furthermore, an agenda-based PSEE could automatically derive the validation obligations arising during a development, and theorem provers could be used to discharge them (if they are expressed formally).

We have defined and used agendas for a variety of software engineering activities that we supported using different formal techniques. These activities include (for more details on the various agendas, the reader is referred to [Hei97]):

- Requirements engineering
  We have defined two different agendas for this purpose. The first supports requirements elicitation by collecting possible events, classifying these events, and expressing requirements as constraints on the traces of events that may occur. Such a requirements description can subsequently be transformed into a formal specification. The second agenda places requirements engineering in a broader context, taking also maintenance considerations into account. This agenda can be adapted to maintain and evolve legacy systems.
- Specification acquisition in general
  There exist several agendas that support the development of formal specifications without referring to a specific application area (such as safety-critical systems). The agendas are organized according to *specification styles* that are language-independent to a large extent.
- Specification of safety-critical software
  Besides the agenda presented in Section 2.1, more agendas for this purpose can be found in [HS97,GHD98].
- Software design using architectural styles
  In [HL97], a characterization of three architectural styles using the formal description language LOTOS is presented. For each of these styles, agendas are defined that support the design of software systems conforming to the style.
- Object-oriented analysis and design
  An agenda for the object-oriented *Fusion* method [CAB+94] makes the dependencies between the various models set up in the analysis and design phases explicit and states several consistency conditions between them.

– Program synthesis
We have defined agendas supporting the development of provably correct programs from first-order specifications. Imperative programs can be synthesized using Gries' approach [Gri81], and functional programs can be synthesized using the KIDS approach [Smi90].

3 The Strategy Framework

In the previous section, we have introduced the agenda concept and have illustrated what kind of technical knowledge can be represented as agendas. Agendas are an informal concept whose application does not depend on machine support. They form the first layer of support for systematic software development.

We now go one step further and provide a second layer with the strategy framework. In this layer, we represent development knowledge *formally*. When development knowledge is represented formally, we can reason about this knowledge and prove properties of it. The second aim of the strategy framework is to support the application of development knowledge by machine in such a way that *semantic* properties of the developed product can be guaranteed.

In the strategy framework, a development activity is conceived as the process of constructing a solution for a given problem. A strategy specifies how to reduce a given problem to a number of subproblems, and how to assemble the solution of the original problem from the solution to the subproblems. The solution to be constructed must be *acceptable* for the problem. Acceptability captures the semantic requirements concerning the product of the development process. In this respect, strategies can achieve stronger quality criteria than is intended, e.g., by CASE. The notion of a strategy is *generic* in the definition of problems, solutions and acceptability.

How strong a notion of acceptability can be chosen depends on the degree of formality of problems and solutions. For program synthesis, both problems and solutions can be formal objects: problems can be formal specifications, solutions can be programs, and acceptability can be the total or partial correctness of the program with respect to the specification. For specification acquisition, on the other hand, we might wish to start from informal requirements. Then problems consist of a combination of informal requirements and pieces of a formal specification. Solutions are formal specifications, and a solution is acceptable with respect to a problem if the combination of the pieces of formal specification contained in the problem with the solution is a semantically valid specification. This notion of acceptability is necessarily weaker than the one for program synthesis, because the adequacy of a formal specification with respect to informal requirements cannot be captured formally. Only if the requirements are also expressed formally, a stronger notion of acceptability is possible for specification acquisition.

The strategy framework is defined in several stages, leading from simple mathematical notions to an elaborated architecture for systems supporting strategy-based problem solving. In the first stages, strategies are defined as a purely declarative knowledge representation mechanism. Experience has shown that formal knowledge representation mechanisms are (i) easier to handle and (ii) have a simpler semantics when they are declarative than when they are procedural. As for strategies, (i) agendas can be transformed into strategies in a routine way (see Section 1), and (ii) the relational semantics of strategies supports

reasoning about and combination of strategies. Further stages gradually transform declaratively represented knowledge into executable constructs that are provided with control structures to guide an actual problem solving process. Figure 4 shows the different stages.

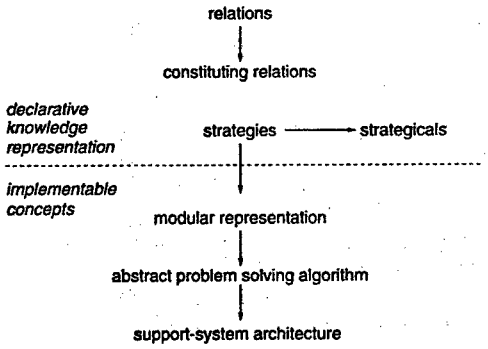


Fig. 4. Stages of definitions

The basic stage consists in defining a suitable notion of *relation*, because, formally, strategies establish a relation between a problem and the subproblems needed so solve it, and between the solutions of the subproblems and the final solution. Relations are then specialized to problem solving, which leads to the definition of *constituting relations*. *Strategies* are defined as sets of constituting relations that fulfill certain requirements. In particular, they may relate problems only to acceptable solutions. *Strategicals* are functions combining strategies; they make it possible to define more powerful strategies from existing ones.

To make strategies implementable, they are represented as *strategy modules*, which rely on constructs available in programming languages. In particular, relations are transformed into functions. The next step toward machine support consists in defining an *abstract problem solving algorithm*. This algorithm describes the manner in which strategy-based problem solving proceeds and can be shown to lead to acceptable solutions. The *generic system architecture* provides a uniform implementation concept for practical support systems.

In the following, we sketch the definitions of the strategy framework (for details, see [Hei97]). Subsequently, we discuss its characteristics. Strategies, strategicals, and strategy modules are formally defined in the language Z [Spi92]. This does not only provide precise definitions of these notions but also makes reasoning about strategies possible.

3.1 Relations

In the context of strategies, it is convenient to refer to the subproblems and their solutions by *names*. Hence, our definition of strategies is based on the the notion of relation as used in the theory of relational databases [Kan90], instead of the usual mathematical notion of relation. In this setting, relations are sets of tuples. A tuple is a mapping from a set of *attributes* to *domains* of these attributes. In this way, each component of a tuple can be

referred to by its attribute name. In order not to confuse these domains with the domain of a relation as it is frequently used in Z, we introduce the type *Value* as the domain for all attributes and define tuples as finite partial functions from attributes to values: $tuple : \mathbb{P}(Attribute \nrightarrow Value)$. Relations are sets of tuples that all have the same domain. This domain is called the *scheme* of the relation.

$$relation : \mathbb{P}(\mathbb{P}\, tuple)$$

$$\forall r : relation \bullet \forall t_1, t_2 : r \bullet \operatorname{dom} t_1 = \operatorname{dom} t_2$$

3.2 Constituting Relations

Constituting relations specialize relations for problem solving. Attributes can either be *ProblemAttributes* or *SolutionAttributes*, whose values must be *Problems* or *Solutions*, respectively. The types *Problem* and *Solution* are generic parameters.

$$const\_rel : \mathbb{P}\, relation$$

$$\begin{aligned}\forall cr : const\_rel \bullet \forall t : cr;\ a : scheme\ cr \bullet \\ scheme\ cr \subseteq (ProblemAttribute \cup SolutionAttribute) \land \\ (a \in ProblemAttribute \Rightarrow t\ a \in Problem) \land \\ (a \in SolutionAttribute \Rightarrow t\ a \in Solution)\end{aligned}$$

Acceptability, the third generic parameter, is a relation between problems and solutions: $\_acceptable\_for\_ : Solution \leftrightarrow Problem$. By default, we use the distinguished attributes *P_init* and *S_final* to refer to the initial problem and its final solution.

The schemes of constituting relations are divided into *input attributes IA* and *output attributes OA*. The constituting relations restrict the values of the output attributes, given the values of the input attributes. Thus, they determine an order on the subproblems that must be respected in the problem solving process. Based on the partitioning of schemes, it is possible to define a dependency relation on constituting relations. A constituting relation cr_2 directly depends on another such relation cr_1 ($cr_1 \sqsubset_d cr_2$) if one of its input attributes is an output attribute of the other relation: $OA\ cr_1 \cap IA\ cr_2 \neq \emptyset$. For any given set *crs* of constituting relations, a *dependency relation* $\sqsubset_{crs}$ is defined to be the transitive closure of the direct dependency relation it determines.

A set of constituting relations defining a strategy must conform to our intuitions about problem solving. Among others, the following conditions must be satisfied:

- The original problem to be solved must be known, i.e. *P_init* must always be an input attribute.
- The solution to the original problem must be the last item to be determined, i.e. *S_final* must always be an output attribute.
- Each attribute value except that of *P_init* must be determined in the problem solving process, i.e., each attribute except *P_init* must occur as an output attribute of some constituting relation.
- The dependency relation on the constituting relations must not be cyclic.

Finite sets of constituting relations fulfilling these and other requirements are called *admissible*. For a complete definition of admissibility, see [Hei97].

Example. For transforming the agenda presented in Section 2.1 into a strategy, we must first define suitable notions of problems, solutions, and acceptability. A problem pr : *SafProblem* consists of three parts: the part $pr.req$ contains an informal requirements description, the part $pr.context$ contains the specification fragments developed so far, and the part $pr.to\_develop$ contains a *schematic* Z-CSP expression that can be instantiated with a concrete one. This schematic expression specifies the syntactic class of the specification fragment to be developed, as well as how the fragment is embedded in its context. Solutions are syntactically correct Z-CSP expressions, and a solution sol : *SafSolution* is acceptable for a problem pr if and only if it belongs to the syntactic class of $pr.to\_develop$, and the combination of $pr.context$ with the instantiated schematic expression yields a semantically valid Z-CSP specification.

3.3 Strategies

We define strategies as admissible sets of constituting relations that fulfill certain conditions. Let $strat = \{cr_0, \ldots, cr_{max}\}$ and $scheme_s\ strat = scheme\ cr_0 \cup \ldots \cup scheme\ cr_{max}$. The set *strat* is a strategy if it is admissible and

- the set $scheme_s\ strat$, contains the attributes *P_init* and *S_final*,
- for each problem attribute a of $scheme_s\ strat$, a corresponding solution attribute, called $sol\ a$, is a member of the scheme, and vice versa,
- if a member of the relation $\bowtie strat$[1] contains acceptable solutions for all problems except *P_init*, then it also contains an acceptable solution for *P_init*. Thus, if all subproblems are solved correctly, then the original problem must be solved correctly as well.

$$
\begin{array}{|l}
strategy : \mathbb{P}(\mathbb{F}\ const\_rel) \\
\hline
\forall strat : strategy \bullet \\
\quad admissible\ strat \land \\
\quad \{P\_init, S\_final\} \subseteq scheme_s\ strat \land \\
\quad (\forall a : ProblemAttribute \bullet a \in scheme_s\ strat \Leftrightarrow sol\ a \in scheme_s\ strat) \land \\
\quad (\forall res : \bowtie strat \bullet \\
\qquad (\forall a : subprs_s\ strat \bullet (res\ (sol\ a))\ acceptable\_for\ (res\ a)) \\
\qquad \Rightarrow (res\ S\_final)\ acceptable\_for\ (res\ P\_init))
\end{array}
$$

The last condition guarantees that a problem that is solved exclusively by application of strategies is solved correctly. This condition requires that strategies solving the problem directly must produce only acceptable solutions. Figure 5 illustrates the definition of strategies, where arrows denote the propagation of attribute values.

Example. When transforming an agenda into a strategy, we must decide which of the steps of the agenda will become subproblems of the strategy. If the result of a step consists in a simple decision or can be assembled from already existing partial solutions, then no

[1] A *join* $\bowtie$ combines two relations. The scheme of the joined relation is the union of the scheme of the given relations. On common elements of the schemes, the values of the attributes must coincide. The operation $\bowtie$ denotes the join of a finite sets of relations.

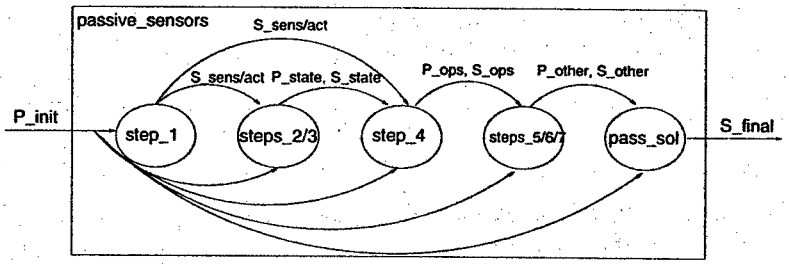


Fig. 5. Strategy for passive sensors

subproblem corresponding to the step is necessary. Considering the agenda of Section 2.1, we decide that Steps 2, 5, and 6 need not become subproblems. Hence, we can define

$$passive\_sensors = \{step\_1, steps\_2/3, step\_4, steps\_5/6/7, pass\_sol\}$$

Figure 5 shows how attribute values are propagated. The constituting relation $step_1$, for example, has as *P_init* as its only input attribute, and *P_sens/act* and *S_sens/act* as its output attributes. The requirements *P_sens/act.req* consist of the requirements *P_init.req* with the addition "Model the sensor values and actuator commands as members of Z types." (see Table 1). The context *P_sens/act.context* is the same as for *P_init*, and *P_sens/act.to_develop* consists of the single metavariable **type_defs** : *Z-ax_def*, which indicates that axiomatic Z definitions have to be developed. For the solution *S_sens/act* of problem *P_sens/act*, the only requirement is that it be acceptable. The other constituting relations are defined analogously. The complete strategy definition can be found in [Hei97].

3.4 Strategicals

Strategicals are functions that take strategies as their arguments and yield strategies as their result. They are useful to define higher-level strategies by combining lower-level ones or to restrict the set of applicable strategies, thus contributing to a larger degree of automation of the development process.

Three strategicals are defined [Hei97] that are useful in different contexts. The THEN strategical composes two strategies. Applications of this strategical can be found in program synthesis. The REPEAT strategical allows stepwise repetition of a strategy. Such a strategical is useful in the context of specification acquisition, where often several items of the same kind need to be developed. To increase applicability of the REPEAT strategical, we also define a LIFT strategical that transforms a strategy for developing one item into a strategy for developing several items of the same kind.

3.5 Modular Representation of Strategies

To make strategies implementable, we must find a suitable representation for them that is closer to the constructs provided by programming languages than relations of database theory. The implementation of a strategy should be a module with a clearly defined interface to other strategies and the rest of the system.

Because strategies are defined as relations, it is possible for a combination of values for the input attributes of a constituting relation to be related to several combinations of values for the output attributes. A type *ExtInfo* is used to select one of these combinations, thus transforming relations into functions. Such external information can be derived from user input or can be computed automatically. A strategy module consists of the following items:

- the set $subp : \mathbb{P}\, ProblemAttribute$ of subproblems it produces,
- a dependency relation $\_depends\_on\_ : ProblemAttribute \leftrightarrow ProblemAttribute$ on these subproblems,
- for each subproblem, a procedure $setup : tuple \times ExtInfo \nrightarrow Problem$ that defines it, using the information in the initial problem and the subproblems and solutions it depends on, and possibly external information,
- for each solution to a subproblem, a predicate $local\_accept : tuple \leftrightarrow Solution$ that checks whether or not the solution conforms to the requirements stated in the constituting relation of which it is an output attribute,
- a procedure $assemble : tuple \times ExtInfo \nrightarrow Solution$ describing how to assemble the final solution, and
- a test $accept\_ : \mathbb{P}\, tuple$ of acceptability for the assembled solution.

Optionally, an *explain* component may be added that explains *why* a solution is acceptable for a problem, e.g., expressed as a correctness proof.

3.6 An Abstract Problem Solving Algorithm

The abstract problem solving algorithm consists of three functions, called *solve*, *apply*, and *solve_subprs*. The function *solve* has a problem *pr* as its input. To solve this problem, a strategy *strat* must be selected from the available strategies. The function *apply* is called that tries to solve the problem *pr* with strategy *strat*. If this is successful, then the value of the attribute *S_final* obtained from the tuple yielded by *apply* is the result of the *solve* function. Otherwise, another trial is made, using a different strategy.

The function *apply* first calls another function *solve_subprs* to solve the subproblems generated by the strategy *strat*. It then sets up the final solution and checks it for acceptability. If the acceptability test fails, *apply* yields a distinguished failure element. Otherwise, it yields a tuple that lies in $\bowtie$ *strat* (see Section 3.3).

The function *solve_subprs* has as its arguments the tuple consisting of the attribute values determined so far, and a set of subproblems still to be solved. It applies *solve* recursively to all subproblems contained in its second argument.

Problem solving with strategies usually requires user interaction. For the functions *solve*, *apply*, and *solve_subprs*, user interaction is simulated by providing them with an additional argument of type seq *UserInput*, where the type *UserInput* comprises all possible user input. User input must be converted into external information, as required by the strategy modules. To achieve this, we use *heuristic functions*. Heuristic functions are those parts of a strategy implementation that can be implemented with varying degrees of automation. It is also possible to automate them gradually by replacing, over time, interactive parts with semi- or fully automatic ones.

It can be proven that the functions *solve*, *apply* and *solve_subprs* model strategy-based problem solving in an appropriate way: Whenever *solve* yields a solution to a problem, then this solution is acceptable.

3.7 Support-System Architecture

We now define a system architecture that describes how to implement support systems for strategy-based problem solving. Figure 6 gives a general view of the architecture which is described in more detail in [HSZ95]. This architecture is a sophisticated implementation of the functions given in the last section. We introduce data structures that represent the state of the development of an artifact. This ensures that the development process is more flexible than would be possible with a naive implementation of these functions in which all intermediate results would be buried on the run-time stack. It is not necessary to first solve a given subproblem completely before starting to solve another one.

Two global data structures represent the state of development: the *development tree* and the *control tree*. The development tree represents the entire development that has taken place so far. Nodes contain problems, information about the strategies applied to them, and solutions to the problems as insofar as they have been determined. Links between siblings represent dependencies on other problems or solutions.

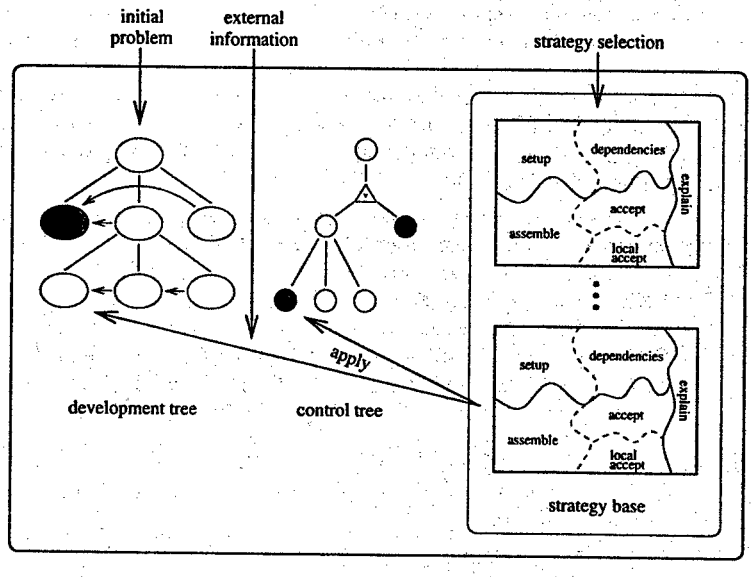


Fig. 6. General view of the system architecture

The data in the control tree are concerned only with the future development. Its nodes represent uncompleted tasks and point to nodes in the development tree that do not yet contain solutions. The degrees of freedom in choosing the next problem to work on are

also represented in the control tree. The third major component of the architecture is the strategy base. It represents knowledge used in strategy-based problem solving via strategy modules.

A development roughly proceeds as follows: the initial problem is the input to the system. It becomes the root node of the development tree. The root of the control tree is set up to point to this problem. Then a loop of strategy applications is entered until a solution for the initial problem has been constructed.

To apply a strategy, first the problem to be reduced is selected from the leaves of the control tree. Secondly, a strategy is selected from the strategy base. Applying the strategy to the problem entails extending the development tree with nodes for the new subproblems, installing the functions of the strategy module in these nodes, and setting up dependency links between them. The control tree must also be extended.

If a strategy immediately produces a solution and does not generate any subproblems, or if solutions to all subproblems of a node in the development tree have been found and tested for local acceptability, then the functions to assemble and accept a solution are called; if the assembling and accepting functions are successful, then the solution is recorded in the respective node of the development tree. Because the control tree contains only references to unsolved problems, it shrinks whenever a solution to a problem is produced, and the problem-solving process terminates when the control tree vanishes. The result of the process is not simply the developed solution – instead, it is a development tree where all nodes contain acceptable solutions. This data structure provides valuable documentation of the development process, which produced it, and can be kept for later reference.

A research prototype that was built to validate the concept of strategy and the system architecture developed for their machine-supported application. The program synthesis system IOSS (Integrated Open Synthesis System) [HSZ95] supports the development of provably correct imperative programs.

3.8 Discussion of Strategies

The most important properties of the strategy framework are:

- **Uniformity.** The strategy framework provides a uniform way of representing development knowledge. It is independent of the development activity that is performed and the language that is used. It provides a uniform mathematical model of problem solving in the context of software engineering.
- **Machine Support.** The uniform modular representation of strategies makes them implementable. The system architecture derived from the formal strategy framework gives guidelines for the implementation of support systems for strategy-based development. Representing the state of development by the data structure of development trees is essential for the practical applicability of the strategy approach. The practicality of the developed concepts is confirmed by the implemented system IOSS.
- **Documentation.** The development tree does not only support the development process. Is also useful when the development is finished, because it provides a documentation of how the solution was developed and can be used as a starting point for later changes.

- **Semantic Properties.** To guarantee acceptability of a solution developed with an implemented system, the functions *local_accept* and *accept* are the only components that have to be verfied. Hence, also support systems that are not verified compeletly can be trustworthy.
- **Stepwise Automation.** Introducing the concept of heuristic function and using these functions in distinguished places in the development process, we have achieved a separation of concerns: the essence of the strategy, i.e. its semantic content, is carefully isolated from questions of replacing user interaction by semi or fully automatic procedures. Hence, gradually automating development processes amounts to local changes of heuristic functions.
- **Scalability.** Using strategicals, more and more elaborate strategies can be defined. In this way, strategies can gradually approximate the size and kind of development steps as they are performed by software engineers.

4 Related Work

Recently, efforts have been made to support re-use of special kinds of software development knowledge: *Design patterns* [GHJV95] have had much success in object-oriented software construction. They represent frequently used ways to combine classes or associate objects to achieve a certain purpose. Furthermore, in the field of software architecture [SG96], *architectural styles* have been defined that capture frequently used design principles for software systems. Apart from the fact that these concepts are more specialized in their application than agendas, the main difference is that design patterns and architectural styles do not describe *processes* but *products.*

Agendas have much in common with approaches to software process modeling [Huf96]. The difference is that software process modeling techniques cover a wider range of activities, e.g., management activities, whereas with agendas we always develop a document, and we do not take roles of developers etc. into account. Agendas concentrate more on technical activities in software engineering. On the other hand, software process modeling does not place so much emphasis on validation issues as agendas do.

Chernack [Che96] uses a concept called *checklist* to support inspection processes. In contrast to agendas, checklists presuppose the existence of a software artifact and aim at detecting defects in this artifact.

Related to our aim to provide methodological support for applying formal techniques is the work of Souquières and Lévy [SL93]. They support specification acquisition with *development operators* that reduce *tasks* to subtasks. However, their approach is limited to specification acquisition, and the development operators do not provide means to validate the developed specification.

Astesiano and Reggio [AR97] also emphasize the importance of method when using formal techniques. In the "method pattern" they set up for formal specification, agendas correspond to *guidelines.*

A prominent example of knowledge-based software engineering, whose aims closely resemble our own, is the Programmer's Apprentice project [RW88]. There, programming knowledge is represented by *clichés,* which are prototypical examples of the artifacts in question. The programming task is performed by "inspection"– i.e., by choosing an

appropriate cliché and customizing it. In comparison to clichés, agendas are more process-oriented.

Wile's [Wil83] development language Paddle provides a means of describing procedures for transforming specifications into programs. Since carrying out a process specified in Paddle involves executing the corresponding program, one disadvantage of this procedural representation of process knowledge is that it enforces a strict depth-first left-to-right processing of the goal structure. This restriction also applies to other, more recent approaches to represent software development processes by process programming languages [Ost87,SSW92].

In the German project KORSO [BJ95], the product of a development is described by a *development graph*. Its nodes are specification or program modules whose static composition and refinement relations are expressed by two kinds of vertices. There is no explicit distinction between "problem nodes" and "solution nodes". The KORSO development graph does not reflect single development steps, and dependencies between subproblems cannot be represented.

The strategy framework uses ideas similar to tactical theorem proving, which has first been employed in Edinburgh LCF [Mil72]. *Tactics* are programs that implement "backward" application of logical rules. The goal-directed, top-down approach to problem solving is common to tactics and strategies. However, tactics set up all subgoals at once when they are invoked. Dependencies between subgoals can only be expressed schematically by the use of *metavariables*. Since tactics only perform goal reduction, there is no equivalent to the *assemble* and *accept* functions of strategies.

5 Conclusions

We have shown that the concept of an agenda bears a strong potential to

- structure processes performed in software engineering,
- make development knowledge explicit and comprehensible,
- support re-use and dissemination of such knowledge,
- guarantee certain quality criteria of the developed products,
- facilitate understanding and evolution of these products,
- contribute to a standardization of products and processes in software engineering that is already taken for granted in other engineering disciplines,
- lay the basis for powerful machine support.

Agendas lead software engineers through different stages of a development and propose validations of the developed product. Following an agenda, software development tasks can be performed in a fairly routine way. When software engineers are relieved from the task to find new ways of structuring and validating the developed artifacts for each new application, they can better concentrate on the peculiarities of the application itself.

We have validated the concept of an agenda by defining and applying a number of agendas for a wide variety of software engineering activities. Currently, agendas are applied in industrial case studies of safety-critical embedded systems in the German project ESPRESS [GHD98].

Furthermore, we have demonstrated that strategies are a suitable concept for the formal representation of development knowledge. The generic nature of strategies makes it

possible to support different development activities. Strategicals contribute to the scalability of the approach. The uniform representation as strategy modules makes strategies implementable and isolates those parts that are responsible for acceptability and the ones that can be subject to automation.

The generic system architecture that complements the formal strategy framework gives guidelines for the implementation of support systems for strategy-based development. The representation of the state of development by the data structure of development trees contributes essentially to the practical applicability of the strategy approach.

In the future, we will investigate to what extent agendas are independent of the language which is used to express the developed artifact, and we will define agendas for other activities such as testing and specific contexts, e.g., object-oriented software development. Furthermore, we will investigate how different instances of the system architecture can be combined. This would provide integrated tool support for larger parts of the software lifecycle.

References

[AR97] E. Astesiano and G. Reggio. Formalism and Method. In M. Bidoit and M. Dauchet, editors, *Proceedings TAPSOFT'97*, LNCS 1214, pages 93–114. Springer-Verlag, 1997.

[BJ95] M. Broy and S. Jähnichen, editors. *KORSO: Methods, Languages, and Tools to Construct Correct Software*. LNCS 1009. Springer-Verlag, 1995.

[CAB+94] D. Coleman, P. Arnold, St. Bodoff, Ch. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1994.

[Che96] Yuri Chernack. A statistical approach to the inspection checklist formal synthesis and improvement. *IEEE Transactions on Software Engineering*, 22(12):866–874, December 1996.

[Dav93] Jim Davies. *Specification and Proof in Real-Time CSP*. Cambridge University Press, 1993.

[GHD98] Wolfgang Grieskamp, Maritta Heisel, and Heiko Dörr. Specifying safety-critical embedded systems with statecharts and Z: An agenda for cyclic software components. In E. Astesiano, editor, *Proc. ETAPS-FASE'98*, 1998. to appear.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, 1995.

[GJ96] P. Garg and M. Jazayeri. Process-centered software engineering environments: A grand tour. In A. Fuggetta and A. Wolf, editors, *Software Process*, number 4 in Trends in Software, chapter 2, pages 25–52. Wiley, 1996.

[Gri81] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[Hei97] Maritta Heisel. *Methodology and Machine Support for the Application of Formal Techniques in Software Engineering*. Habilitation Thesis, TU Berlin, 1997.

[HL97] Maritta Heisel and Nicole Lévy. Using LOTOS patterns to characterize architectural styles. In M. Bidoit and M. Dauchet, editors, *Proceedings TAPSOFT'97*, LNCS 1214, pages 818–832. Springer-Verlag, 1997.

[HS96] Maritta Heisel and Carsten Sühl. Formal specification of safety-critical software with Z and real-time CSP. In E. Schoitsch, editor, *Proceedings 15th International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, pages 31–45. Springer-Verlag London, 1996.

[HS97] Maritta Heisel and Carsten Sühl. Methodological support for formally specifying safety-critical software. In P. Daniel, editor, *Proceedings 16th International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, pages 295–308. Springer-Verlag London, 1997.

[HSZ95] Maritta Heisel, Thomas Santen, and Dominik Zimmermann. Tool support for formal software development: A generic architecture. In W. Schäfer and P. Botella, editors, *Proceedings 5-th European Software Engineering Conference*, LNCS 989, pages 272–293. Springer-Verlag, 1995.

[Huf96] Karen Huff. Software process modelling. In A. Fuggetta and A. Wolf, editors, *Software Process*, number 4 in Trends in Software, chapter 2, pages 1–24. Wiley, 1996.

[Kan90] Paris C. Kanellakis. Elements of relational database theory. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 17, pages 1073–1156. Elsevier, 1990.

[Mil72] Robin Milner. Logic for computable functions: description of a machine implementation. *SIGPLAN Notices*, 7:1–6, 1972.

[Ost87] Leon Osterweil. Software processes are software too. In *9th International Conference on Software Engineering*, pages 2–13. IEEE Computer Society Press, 1987.

[RW88] Charles Rich and Richard C. Waters. The programmer's apprentice: A research overview. *IEEE Computer*, pages 10–25, November 1988.

[SG96] Mary Shaw and David Garlan. *Software Architecture.* IEEE Computer Society Press, Los Alamitos, 1996.

[SL93] Jeanine Souquières and Nicole Lévy. Description of specification developments. In *Proc. of Requirements Engineering '93*, pages 216–223, 1993.

[Smi90] Douglas R. Smith. KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.

[Spi92] J. M. Spivey. *The Z Notation – A Reference Manual.* Prentice Hall, 2nd edition, 1992.

[SSW92] Terry Shepard, Steve Sibbald, and Colin Wortley. A visual software process language. *Communications of the ACM*, 35(4):37–44, April 1992.

[Wil83] David S. Wile. Program developments: Formal explanations of implementations. *Communications of the ACM*, 26(11):902–911, November 1983.

Requirements Engineering Repositories: Formal Support for Informal Teamwork Methods

Hans W. Nissen
RWTH Aachen
Informatik V
Ahornstr. 55
52072 Aachen, Germany
nissen@informatik.rwth-aachen.de

Matthias Jarke
RWTH Aachen
Informatik V
Ahornstr. 55
52072 Aachen, Germany
jarke@informatik.rwth-aachen.de

Abstract

Relationships among different modeling perspectives have been systematically investigated focusing either on given notations (e.g. OMT) or on domain reference models (e.g. SAP). In contrast, many successful informal methods for business analysis and requirements engineering (e.g. JAD) emphasize team negotiation, goal orientation and flexibility of modeling notations. This paper addresses the question how much formal and computerized support can be provided in such settings without destroying their creative tenor. Our solution comprises four components:

(1) A modular conceptual modeling formalism organizes individual perspectives and their interrelationships. (2) Perspective schemata are linked to a conceptual meta meta model of shared domain terms, thus giving the architecture a semantic meaning and enabling adaptability and extensibility of the network of perspectives. (3) Inconsistency management across perspectives is handled in a goal-oriented manner, by defining the analysis goals as meta rules which are automatically adapted to perspective schemata. (4) Continuous incremental maintenance of inconsistency information is provided by exploiting recent view maintenance techniques from deductive databases.

The approach has been fully implemented as an extension to the ConceptBase meta database management system and is currently experimentally applied in the context of business analysis and data warehouse design.

1 Introduction

As observed in [Poh94], modeling processes proceed along three dimensions: representational transformation, domain knowledge acquisition, and stakeholder agreement. Existing methodologies tend to emphasize one of these dimensions over the others: the modeling *notations*, the available *knowledge* within a specific domain, or the *people* involved in the analysis project. All three method types have a long history, with little interaction between them.

The management of inconsistent partial models is an inevitable part of requirements engineering (RE) [Bal91, FF91, Eas96]. Multiple stakeholders with conflicting opinions, contradicting requirements and alternative perspectives cause these inconsistencies.

All methodologies support multiple partial models to represent the set of requirements. They differ substantially in the preferred coupling of the partial models. This requires cross-model analysis to guarantee consistency between the partial models. The extent, justification and customizability of the performed analysis constitute a main difference between the reviewed methodologies and illustrate the specific problems with the team- and goal-oriented methods.

Notation-oriented methods manifest their assistance in the set of modeling notations they offer. Their philosophy can be characterized by the slogan *In the language lies the power.* Examples of notation-oriented methods are structured analysis approaches, as, e.g., Modern Structured Analysis [You89], and object-oriented techniques, as, e.g., UML [FS97]. They attach great importance to the consistency of the developed models. Conflicts between stakeholders, inconsistencies induced by a different terminology or simple name clashes have to be resolved 'outside' the model. The analysis goals are defined directly on the constructs of the notations, i.e. on the contents of the corresponding (maybe hypothetical) meta models.

Since the mid-80s, researchers have attempted to formalize semi-formal notations via a transformation to well-understood specification formalisms like logic [Gre84], graph grammars and algebraic specifications. They specify a fixed semantics the user must accept and cannot modify. A recent and very comprehensive example is a formal semantics for SSADM (Structured Systems Analysis and Design Method) [DCC92] based on algebraic specification [BFG+93] by [Hus94].

A different strategy is employed by the **domain-oriented analysis methods**. For a specific application domain, e.g., public administration or furniture industry, they offer a predefined set of reference models. Reference models describe typical data, processes and functions, together with a set of consistency tests which evaluate relationships between the models. Reference models represent the knowledge collected in multiple analysis projects within a particular domain: *In the knowledge lies the power.* The reuse of reference models can considerably reduce the analysis effort. However, it can be inflexible since the user can tailor the notations, the constraints or contents only to the degree foreseen by the developers of the reference models.

The SAP Business Blueprints are reference models from the business domain [SAP97]. The Aris Toolset [IDS96] offers a platform for working with reference models. It offers hard-coded constraint checks within and across the models. Analysis goals exist only

implicitly. They are reflected by the implementation of the reference models, i.e. by the contents and the structure of the models.

Goal- and team-oriented approaches specifically address the objective to capture requirements from different information sources and to make arising conflicts productive: *In the people lies the power.* Prominent examples include IBM's JAD (Joint Application Design) [Aug91], SSM (Soft Systems Methodology) [Che89], and PFR (Analysis of Presence and Future Requirements) [Abe95] focus on the *early* and *direct* participation of all stakeholders, the rapid generation of *joint results*, the *tolerance* of conflicting perspectives, a *goal-oriented process*, and *informal, graphical notations*. Experiences in the application of JAD give evidence for a 20% to 60% increased productivity compared with semi-formal and formal methods [GJ87, Cra94]. It is typical for these methods that the execution is supported by highly skilled group facilitators which animate the participants, guide the analysis process and keep an eye on the compliance with the specified analysis goals. Conflicts and inconsistencies are tolerated for the benefit of a fast and creative acquisition process. Moreover conflicts are employed as a tool for the analysis process. For each topic in the domain they collect perspectives from different stakeholders to provoke conflicts and use them to guide further discussions and interviews.

Teamwork remains very informal to enhance creativity. Neither notations nor analysis goals are predefined by the methods but specified by the participants according to the actual problem to be solved. To accomodate the change of goals during project execution, the customization of analysis goals and notations is possible even during a running project. At present, no supporting tools are available beyond simple groupware tools. The main reasons for this dilemma are the high degree of customizability the tools must offer and the lack of formalizations available.

In the next section we give a detailed overview of our approach. In sections 3 and 4 we then present the main contributions. The axiomatization of a modular conceptual modeling language yields a formally precise way of how to define perspectives and control the information exchange between them. On top of this basic formalism, we develop techniques for the definition, compilation, and distribution of analysis goals in a customizable modeling environment (i.e. one without a fixed set of notations or domain models). A distributed execution environment enables efficient incremental maintenance of instance-level information about violations of analysis goals. The paper ends with a comparison to related work in section 5 and a summary and outlook in section 6.

2 Overview of the Approach

The informal team-based information acquisition is not subject to any formalization (though it could at least be recorded in multimedia). The potential for formalization and computer-based tools lies in the other part of the methods: the *cross-perspective analysis* which is performed by moderators and analysts to extract knowledge and further questions from the collected information. The team produces many perspectives in a very short time containing lots of conflicts and inconsistencies. The situation for the analysts becomes even harder since the notations and the analysis goals can change from

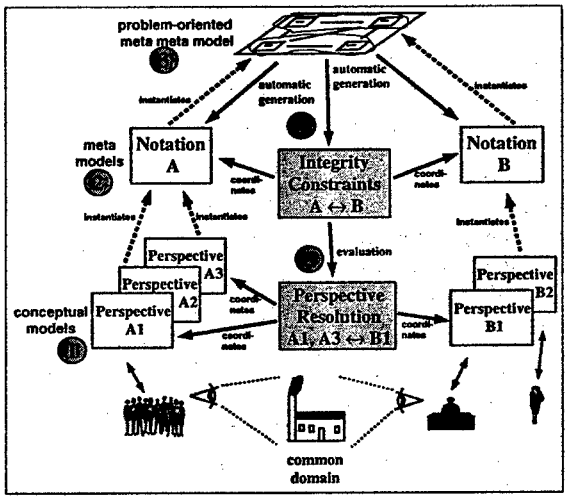


Figure 1: Overview of the Approach

one project to another. The manual comparison of perspectives becomes a big problem and is both time-consuming and error-prone.

2.1 Components

The basis of our formalization are the analysis goals and the notations as specified by the stakeholders at the beginning of method execution. We forge links to the notation- and the domain-oriented methods by formally transforming the domain-oriented analysis goals into integrity constraints over notational meta models. The result is a set of syntactic and domain-oriented inter-relationships between the notations, just as it is the case in the notation- and the domain-oriented methods. But in our approach they are automatically generated from user-defined declarative, notation-independent specifications of analysis goals. Figure 1 presents the components of our approach.

(1): Separation of Multiple Perspectives. The conceptual models represent individual perspectives of stakeholders. The figure shows three perspectives (`A1,A2,A3`) expressed in `Notation A` and two perspectives (`B1,B2`) expressed in `Notation B`. Our separation mechanism offers independent modeling contexts (modules) and enables the representation of inconsistent conceptual models.

(2): Extensible Meta Modeling. The notations used to express the perspectives are defined on the second level, the meta level. The example comprises two notations, `Notation A` and `Notation B`. Since the notations are subject to modification, the modeling formalism must support the creation and modification of meta models. These meta models also reside in separate modules.

(3): Specifying Common Domain Terms and Analysis Goals. A shared meta meta model inter-relates all perspective notations. It specifies the domain structure and the analysis goals. This model is created by teamwork at the beginning of the analysis project and documents the common language of all participating stakeholders. The perspective notations are views on this model, i.e. a notation covers a specific fragment of the common domain terms.

This assignment of notations to domain fragments gives the architecture a semantic meaning. The semantics of the domain structure is defined by the analysis goals - formulas which formalize the correct or intended behaviour as well as expected problems of the domain components. These goals define the scope of the cross-perspective check performed on the bottom level between concrete conceptual models.

(4): Transformation of Analysis Goals. To close the gap between the domain-oriented analysis goals and the perspectives expressed in notations and to enable a distributed modeling activity and consistency check, the analysis goals of the meta meta model are automatically transformed to integrity constraints on the notation meta models. The relationships between different notations reside in so-called resolution modules which are connected to the corresponding notation modules via `coordinates` links. In the figure we have a resolution module for the two notations mentioned before.

(5): Continuous Inconsistency Documentation. To avoid interrupting the creative modeling activity in the presence of inter-perspective inconsistencies, the cross-perspective analysis takes place in separate resolution modules. The figure presents a resolution module (the shaded module on the bottom level) to check the perspectives `A1`, `A3` and `B1`. This is also the place where the conflicts are documented and continuously monitored.

2.2 Industrial Application: Supporting the PFR Analysis Method

In [NJJ+96] we reported the application of our approach to the PFR analysis method. We use this application as a running example.

PFR is mainly employed in the early phases of projects developing information systems supporting business processes. The method has three steps:

- In a two-day workshop, stakeholders agree on the scope of the analysis project: the current problems which should be solved and in correspondence to this, the domain structure and the analysis goals. The group also makes a rough analysis of the current business processes in terms of information exchange among organizational units, identifies weak spots and drafts a redesigned business process.
- The perspectives identified as critical to success are then captured in detail by interviews, workflow and document analysis. The acquisition process is accompanied by a *cross-perspective analysis* of the captured information for consistency, completeness, and local stakeholder agreement. The results of the comparisons guide subsequent interviews to clarify conflicts and complete the models.
- In a second workshop the goal is to draw together individual perspectives to achieve global *stakeholder agreement*. The step is accompanied and followed by the development of a comprehensive requirements document of typically several hundred pages.

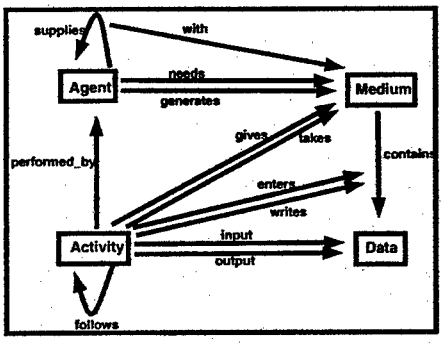


Figure 2: Example: The PFR meta meta model

The first workshop leads to a problem-oriented meta meta model defining shared domain terms and analysis goals. Figure 2 presents a model that is used as a default in PFR analysis projects. It is modified to fit to the actual problems and analysis goals if necessary. The current status of the processes is analysed from three perspectives: the *information-exchange* within the first workshop, the *activity-sequence* and the *document-structure* within the detailed acquisition process in the second step.

The meta meta model in figure 2 explains the basic concepts of these perspectives and their interrelationships. The information-exchange perspective is represented by an `Agent` who `supplies` other agents `with` a `Medium`, the activity-sequence by the `Activity` that is `performed_by` an `Agent` and produces `Data` as `input` or `output`, and the document-structure by a `Medium` that `contains Data`.

The meta meta model contains a precise description of the terms that are employed during a PFR analysis. Its structure focuses on the expected problems in the specific domain. The distinction between `Medium` and `Data`, for example, is essential to talk about the unnecessary exchange of documents, i.e. documents which contain data that is not used by any activity.

Figure 3 presents a part of the PFR environment as an example of the two top levels of our architecture. The top level module contains the PFR meta meta model together with an analysis goal stating that "Every exchanged Medium must contain Data". This goal formalizes the basic requirement that an efficient business process should not include exchange of documents that do not contain useful data. The formal definition of this goal will be given in section 4.

Below the PFR meta meta model reside the notation meta models to formulate the information-exchange and the document-structure perspectives. The resolution module connected to both notation modules contains the transformed version of the above analysis goal. It specifies an integrity constraint on both perspective notations. The exact formal definition of this integrity constraint will also be given in section 4.

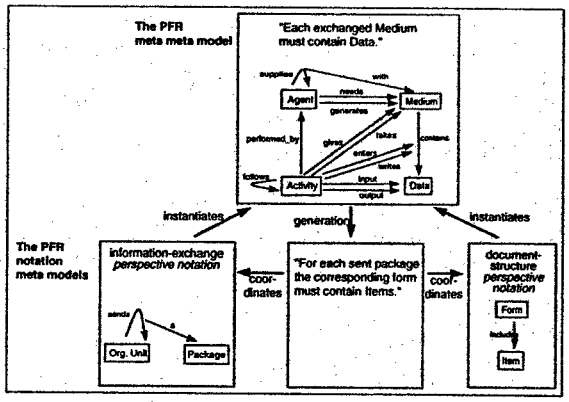


Figure 3: Example: The PFR resolution architecture

3 M-Telos: Separation of Multiple Perspectives

The conceptual modeling language Telos [MBJK90] was designed for managing (meta) information about information systems. It integrates aspects from database design, knowledge representation and conceptual modeling. The object-oriented Telos data model supports the abstraction principles classification/instantiation, specialization/generalization and aggregation. A basic concept of Telos is the representation of every single piece of information as an object with its own identity. The unlimited instantiation hierarchy enables classes to be themselves instances of other classes, so-called meta classes. Meta classes may again be instances of meta meta classes, and so on.

A version of Telos called O-Telos was formalised in [Jeu92] and implemented in ConceptBase [JGJ+95], a deductive object manager for meta databases. This axiomatization enables the interpretation of an Telos object base as a special case of a deductive database with stratified negation and perfect model semantics [Min87].

M-Telos extends O-Telos by introducing so-called **modules** as a separation mechanism. A module provides an independent modeling context where users can create an individual analysis perspective in the form of a conceptual model. The intended application scenario of modules in concurrent conceptual modeling processes induces the need for **communication** between modules [NKF94]. The module concept supports cooperation among group members by the possibility to define local modules.

It is often the case that one modeling task depends on another one and reuses a part of its results. To support this situation, two modules can communicate by setting up an **import-relationship**. The importing module obtains access to the contents of the imported module. To protect a specific part of the module contents, the concept allows the division of the accessible contents of a module into a private and a public part.

We need not only a modeling context but also a context for the resolution of multiple perspectives. We use dedicated modules for this monitoring task, the so-called resolution modules (cf. section 4). As our experiences indicate [NJJ+96], such resolution modules

need a special way to access the monitored modules. Therefore the module concept offers a **coordination relationship** between modules which enable a resolution module to access all accessible objects of the monitored modules.

3.1 Formal Definition of M-Telos

The semantics of a M-Telos object base is given by a mapping to a deductive database containing predefined objects, integrity constraints and deductive rules. This set of objects, constraints and rules constitute the **axiomatization** of M-Telos.

The basic data structure of M-Telos is very simple. It represents all information using labeled nodes and arcs with object identity.

Definition 3.1 (Extensional Object Base)
Let ID be a set of object identifiers, LAB be a set of labels. An **extensional object base** OB is defined as a finite set of objects:

$$OB \subseteq \{ P(o,s,l,d) \mid o,s,d, \in ID,\ l \in LAB \}.$$

Every object is represented in form of a tuple $P(o,s,l,d)$ with object identifier o, start object s, destination object d and label l. The above object o can be read as: "The object s has a relationship called l to the object d". We distinguish four different categories of objects: Objects of the form $P(o,o,l,o)$ are called **individuals**. They represent self-standing entities. Objects containing the special label *in* like $P(o,s,in,d)$ describe **instantiation relationships**. Objects containing the label *isa* like $P(o,c,isa,d)$ represent **specialization relationships**. All other objects denote just **attributes**.

Five predefined objects document the above mentioned four object categories: *Object* contains all objects of an extensional object base as instances; *Individual*, $InstanceOf$, IsA, and *attribute* contain the individuals, instantiation, specialization and attribute relationships as instances (cf. axioms A-1 to A-5).

M-Telos introduces an additional predefined object called *Module* which contains all modules as instances. It offers four attributes: *contains* links to the objects which are defined within the specific module; *exports* declares accessible objects to be public; $imports\_from$ refers to another module and indicates an import relationship; *coordinates* indicates a coordination relationship to another module (cf. axioms A-6 to A-10). A special predefined module called *System* contains all predefined objects including itself (cf. axioms A-11.1 to A-11.26).

Figure 4 visualizes the predefined objects as a semantic network. The attributes specifying the contents of *System* are omitted for readability. Individual objects are denoted as nodes of the graph, instantiation, specialization and attribute relationships as dotted, shaded and labelled directed arcs between their source and destination components.

Due to space limitations, we concentrate on the axioms that define the properties of the module concept. The complete list of axioms can be found in appendix A [Nis96].

M-Telos requires that the names of individual objects must be unique among all individuals defined within the same module (A-16). Note that names need not be unique

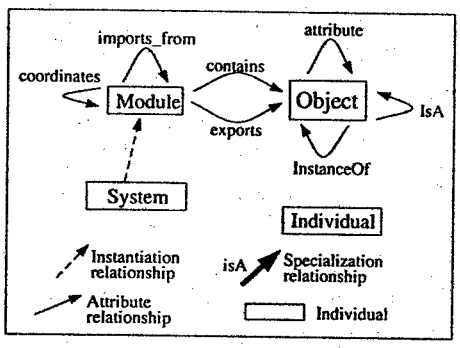


Figure 4: Predefined objects in M-Telos

among all objects that are *accessible* in a module. Within a single module two individuals with the same name but different object identifiers and different defining modules may exist. Especially for the representation of multiple conceptual models which may be developed by different people observing the same domain this is an essential feature.

The intention behind the module concept is that the access to objects is only possible through modules, i.e. every object should belong to exactly one module. Axiom A-18 requires a defining module for every object of an extensional object base. In addition axiom A-19 requires that every object is contained in only one module.

Until now there was no possibility to talk about the set of accessible objects within a module (as opposed to defined). The new literal $P^{Mod}(M, o, s, l, d)$ describes the objects $P(o, s, l, d)$ which are accessible in module M. The set of accessible objects for a module M comprises

- all objects defined within M (axiom A-22),
- all imported objects (axiom A-23),
- all accessible objects of coordinated modules (axiom A-24), and
- all objects that are accessible in the containing module (axiom A-25).

The closed world assumption (CWA) [Min87] guarantees that exactly the literals $P^{Mod}(M, o, x, l, y)$ that can be deduced using these rules hold and no others.

The exported objects of a module must form a subset of all the accessible objects of that module. Axiom A-48 formulates this as an integrity constraint. The export part describes a subpart of the conceptual model formed by all accessible objects. This subpart is exported to be reused and extended in another module. To be able to reuse the exported subpart all referenced objects must also be included. This requirement of referential integrity particularly for the export part is formalised in axiom A-49.

For the perspective resolution we shall need the *coordinates* relationship between two modules. The coordination relation is transitive, i.e. if module $M1$ coordinates module $M2$ and $M2$ coordinates module $M3$ then $M1$ also coordintes (indirectly) the module $M3$. This fact is expressed in axiom A-50. Since a cyclic coordination relationship leads

to enormous problems in the handling of perspective inter-relationships (see section 4), we explicitly forbid this with axiom A-51.

On basis of the axioms presented so far, other properties of M-Telos are defined which are not directly related to the module concept. We mention in the following only the important ones.

- *Instantiation axiom (A-44):* An instance of a class is allowed to instantiate the class's attributes.
- *Specialization axiom (A-43):* The destination (called superclass) of a specialization relationship inherits all instances of the source (called subclass).

  In combination with the instantiation axiom this defines the attribute inheritance from superclass to subclass: instances of the subclass can instantiate attributes of the superclass.
- *Multiple generalization/instantiation axiom (A-45, A-47):* M-Telos supports multi-classification and multi-generalization under some restrictions.
- *System classes axioms (A-29 to A-39):* For every object the instantiation relationships to the predefined objects *Object*, *Individual*, $InstanceOf$, IsA and *attribute* are deduced and may not be contained in the extensional object base. Also, every instance of the objects *Individual*, $InstanceOf$, IsA and *attribute* must have the specific structure introduced at the beginning of this section.

3.2 Properties of the Axiomatization

A main goal of the axiomatization of M-Telos was to preserve the **simplicity** of the O-Telos formalization given in [Jeu92]. We formulated 76 axioms, of which 32 were slightly modified axioms from the O-Telos formalization. 31 of the new axioms are new predefined objects. This number results from the definition of the module *System*. Otherwise we only defined seven new rules and six new constraints.

Now we are able to formally define a consistent M-Telos object base. It forms a special deductive database. A deductive database is a triple (EDB, IDB, IC) where EDB, the *extensional database* is a set of facts in the form of relations, IDB, the *intensional database*, is a set of deductive rules defining intensional relations, and IC is a set of closed formulas stating integrity constraints. For a M-Telos object base, EDB becomes the extensional object base containing only facts of the P-relation, IDB is exactly the set of axioms forming deductive rules, and IC is exactly the set of the axioms interpreted as integrity constraints.

Definition 3.2 (M-Telos Object Base)
Let AX_{OB} be all axioms describing predefined objects, AX_R be all axioms which are deductive rules and AX_{IC} be all axioms which are constraints.

Then the triple (OB, AX_R, AX_{IC}) is a **M-Telos object base** if $AX_{OB} \subseteq OB$ holds.

(OB, AX_R, AX_{IC}) is called a **consistent M-Telos object base** if the perfect model of (OB, AX_R) satisfies all integrity constraints of AX_{IC}.

An important property of an object base is the referential integrity within each module. This property guarantees that for every accessible object in a module also the destination and the source components are accessible objects in that module. The following proposition proves this property for consistent M-Telos object bases.

Proposition 3.3 (Referential Integrity [Nis96])
Let (OB, R, IC) be a consistent M-Telos object base. Then for every object $P(o, s, l, d) \in OB$ and every module M with $P(\#M, \#M, M, \#M) \in OB$ and $In(\#M, \#Module)$ deducible from OB holds: If the object o is accessible in M, i.e. $P^{Mod}(\#M, o, s, l, d)$ is deducible from OB using the rules from R, then also the objects with the identifiers s and d are accessible in M.

The following propositions formalizes the architecture of a modular knowledge base. The modules always form a tree such that the contents of the System module will be accessible in every single module.

Proposition 3.4 (*contains* Relation Forms a Unique Tree [Nis96])
Let (OB, R, IC) be a consistent M-Telos object base. Then (a) all modules are directly or indirectly contained in *System* and (b) the *contains* relation forms a tree.

A distinctive feature of O-Telos is the possibility for unlimited metamodeling. It allows the user to build meta models, meta meta models and so on. This property is preserved in M-Telos. Meta modeling is still possible without any restrictions within a module. In addition the modules can be arranged according to their degree of abstraction. A module then contains only conceptual models of one abstraction level, the more concrete level and the more abstract level reside then in different modules. Of course, any combination of these approaches is also possible.

4 Goal-oriented and Customizable Inconsistency Management

The goal- and team-oriented methods are designed to produce the maximum of conflicts possible. They employ highly overlapping perspectives and acquire almost all information from different stakeholders (cf. section 1). Due to this redundancy, a large number of relationships exist between perspectives. The cross-perspective analysis checks these relationships. It follows the goals specified in the problem-oriented meta meta model.

The analysis goals are formulated as so called **meta formulas**: They make statements about objects that reside two abstraction levels below the formula. This is necessary since the perspectives we actually want to analyse reside two levels below the meta meta model (cf. figure 1).

In more technical words: A formula φ is called a meta formula if φ contains a literal (a) $In(x, c)$ where c is a variable, or (b) $A(x, m, y)$ where x is not range restricted by a literal $In(x, c)$ where c is a constant. In such a case we call these literals meta literals.

Example 4.1 (Meta Formula)
The following meta formula is the formalization of the analysis goal given in figure 3 in natural language. It contains several meta literals as, e.g., $In(med, m'), In(supp, s')$, $In(with, w')$. The variables $med, supp, with$ denote objects two levels below the meta meta model. They are not bound to any concrete object on the meta level; m', s', w' are again variables.

$$\begin{aligned} &In(m', \#Medium) \wedge In(d', \#Data) \wedge In(s', \#Agent!supplies) \wedge \\ &In(e', \#Medium!contains) \wedge In(w', \#Agent!supplies!with) \wedge In(med, m') \wedge \\ &In(supp, s') \wedge In(with, w') \wedge From(with, supp) \wedge To(with, med) \\ &\Rightarrow \exists\, data, cont\ In(data, d') \wedge In(cont, e') \\ &\wedge From(cont, med) \wedge To(cont, data) \end{aligned}$$

□

Analysis goals for the PFR method exist for a single perspective, for dependencies between multiple perspectives, and to test the desirability of the modeled business processes. Although it is possible to use the analysis goals as they are we will transform them to integrity constraints on the notations's meta models. At this point we have to make clear our terminology in the following subsections: an *analysis goal* is a formula that is specified within the meta meta model and is thus a meta formula. An *integrity constraint* is a formula that is not a meta formula.

The analysis goals represent the agreement among the stakeholders about the goals, or more specific, the questions and problems, the analysis project is dedicated to. Accordingly, our architecture manages the analysis goals within the central module. But the analysis and modeling process does not run in a centralizes way, it is distributed and involves many agents. A central control instance will then be a system bottleneck. On the other hand, a complete distribution without any central control instance like in [NKF94] would not cover the global relationships. To be efficient in such a setting and at the

same time be able to manage the global connection we follow the approach of [ACM95] which is a compromise of the two extremes mentioned before: The environments for the perspective development are distributed and work autonomously but there still exists a central instance which has knowledge about their possible inter-relationships. Applied to our case: from the global meta meta model we generate integrity constraints which can be evaluated locally within the modules of the meta level. The global module needs not be accessed during inconsistency monitoring time.

We explain the mechanism in two steps: First we describe the technique of partial evaluation which is used to transform an analysis goal into integrity constraints. Since not all analysis goals need to be transformed to all modules we guide the partial evaluation by generating a transformation plan, i.e. the assignment of analysis goals to modules. The algorithms to compute these plans are subject of the second step.

The last subsection gives then a short overview of the continuous inconsistency management on the instance level. More details on this part are given in [NJ97].

4.1 Partial Evaluation of Analysis Goals

We employ the technique of partial evaluation to transform a meta formula into an integrity constraint. In [Jeu92] the application of this technique to the O-Telos object model is presented. We can directly adapt the results to M-Telos. We will therefore only sketch the technique of partial evaluation.

A meta formula contains (meta) literals $In(x, c)$ where c is not a constant but a variable. In our specific case this variable is used to denote an object of the meta model of a notation. The typical situation is thus to have two such literals - $In(x, y)$ and $In(y, c)$ - within one formula where c is an object of the meta meta model and x and y are variables. The meta formula then makes statements about the behaviour of x which denotes an object of a conceptual model. The goal of partial evaluation is to find a solution for y by evaluating the literal $In(y, c)$ within a specific module of the object base. Each occurence of y within the meta formula is then replaced by the computed object. In our case this object comes from the meta level and denotes an object of the meta model of a notation. Since we then already know that the literal $In(y, c)$ evaluates to true with the computed object we can omit the literal and simplify the formula. For every solution of that literal we get a new, partially evaluated version of the original meta formula.

This process has to be repeated for every meta literal until all meta literals have been evaluated. Since we evaluate a meta formula always within a specific module we do it on basis of a notations meta model or a resolution module. The resulting formulas contain no more objects of the meta meta model but only objects of the meta model of a notation. It is therefore only valid for that notation. If not all meta literals of a meta formula can be evaluated in a module then this meta formula is not partially evaluable within this specific module.

Example 4.2 (Transformed Meta Formula)
The following formula is the transformed version of the meta formula presented in example 4.1. The meta variables are replaced by concrete objects of the two notations for information-exchange and document-structure perspectives (cf. figure 3). The literal $In(med, m')$ of the meta formula is replaced by $In(med, \#Package)$. In addition all the literals connecting these variables to objects of the meta meta model as, e.g., $In(m', \#Medium)$, are eliminated.

$$\begin{aligned} &In(med, \#Package) \wedge In(supp, \#OrgUnit!sends) \wedge In(with, \#OrgUnit!sends!a) \wedge \\ &\quad From(with, supp) \wedge To(with, med) \\ &\Rightarrow \exists\, data, cont\ In(data, \#Item) \wedge In(cont, \#Form!includes) \\ &\quad \wedge From(cont, med) \wedge To(cont, data) \end{aligned}$$

□

4.2 Computation of Transformation Plans

Unnecessary partial evaluations of analysis goals may arise if modules are connected via *coordinates* relationships: If there exists a *coordinates* link from module A to module B then everything accessible in B is also accessible in A. Any analysis goal that could be transformed for B therefore can also be transformed for A. Since the transformed constraint becomes accessible in A anyway, a separate transformation for A is not necessary. To avoid such inefficiency we compute for every analysis goal the minimal set of modules it must be transformed to.

Algorithm 4.1 (Computation of Destination Modules)
The algorithm first computes the following sets and functions:

- the set M of all modules of the meta level
  $M = \{M_1, \ldots, M_n\}$,
- the set C of all specified *coordinates* relationships:
  $C = \{(M_{1,1}, M_{1,2}), \ldots, (M_{m,1}, M_{m,2})\}$, where $(M_{i,1}, M_{i,2})$ denotes a *coordinates* relationship from $M_{i,1}$ to $M_{i,2}$.
  The set C denotes a directed, acyclic graph.
- the set AG of all analysis goals specified within the meta meta model:
  $AG = \{\varphi_1, \ldots, \varphi_l\}$,
- the function *applicable* which computes for each analysis goal $\varphi \in AG$ the set of modules for which a partial evaluation is possible:
  $applicable : AG \rightarrow \wp(M)$
  Whether $M \in applicable(\varphi)$ holds or not is computed by a comparison of the quantifications in φ with the instantiation relationships of M to the meta meta model. Only if for every quantification in φ an instantiation in M exists all meta literals in φ can be evaluated.

For every analysis goal $\varphi \in AG$
  compute $P^{\varphi} = applicable(\varphi)$, the set of potential destination modules
  compute the set C^{φ} with
    $C^{\varphi} = \{(M_{i,1}, M_{i,2}) : (M_{i,1}, M_{i,2}) \in C, M_{i,1} \in P^{\varphi}, M_{i,2} \in P^{\varphi}\}$
  let $E^{\varphi} = P^{\varphi}$
  for each $(M_{i,1}, M_{i,2}) \in C^{\varphi}$
    do
    delete $M_{i,1}$ from E^{φ}
    od

□

The resulting set E^{φ} denotes all the modules of the meta level for which a transformation of φ is necessary. The set E^{φ} is independent from the selections out of C^{φ} and is always uniquely determined. For E^{φ} we can prove correctness and completeness concerning the accessibility of transformed integrity constraints as well as its minimality.

Proposition 4.3 (Correctness and Completeness)
The computed set E^{φ} for an analysis goal φ is

(a) correct, i.e. E^{φ} contains only such modules for which a partial evaluation of φ is allowed.

(b) complete, i.e. the transformation of φ with respect to all modules in E^{φ} results in the accessibility within all potential destination modules.

Proposition 4.4 (Minimality)
For every analysis goal φ the algorithm 4.1 computes in E^{φ} the minimal set of destination modules such that its accessibility in all potential destination modules is guaranteed.

The above proved completeness is defined with respect to the given module structure on the meta level. But there exists a second view on completeness with respect to the analysis goals of the meta meta model: The transformation is complete if all analysis goals of the meta meta model have been transformed.

This kind of completeness does not always hold. If there is no notation for a specific fragment of the meta meta model then the analysis goals specified for this fragment could not be transformed to integrity constraints. The result is that some formal statements represented within the meta meta model could not be tested during the analysis process. In some cases this kind of incompleteness is not a problem or even desired. But in all cases it is useful for the users to get information about analysis goals that can not be transformed.

An automatic completion of the module structure on the meta level is not always possible. A tool could constitute additional resolution modules but cannot automatically establish new notations if a fragment of the meta meta model is not covered yet. We developed an algorithm computing additional resolution modules such that an analysis goal becomes transformable (see [NJ97] for details). The result is in general not minimal. One minimal set can be computed by an algorithm which follows the computation of a minimal set of functional dependencies in relational database schema design [EN94]: A module is eliminated if its relationships are covered by the reminding modules in the set. We present such an algorithm in [NJ97].

4.3 Continuous Inconsistency Management

Inconsistencies detected by goal-oriented inconsistency management are often not repaired immediately. Analysts must continue their current work without the need to check their model for syntactic correctness or to resolve conflicts with other users. The process of the goal- and team-oriented methods offer special meetings and interviews where the list of detected inconsistencies are discussed among the users. Due to space limitations we only sketch our inconsistency management approach. The detailed presentation is given in [NJ97].

We employ view maintenance technology [Sta96] for the continuous inconsistency management. We rewrite integrity constraints as deductive view definitions such that the view contents represents constraint violations. Inconsistency monitoring is then achieved by monitoring such *inconsistency views*. We capture the reasons behind integrity violations by recomputing the derivation trees using a meta interpreter. In the continuous version we declare objects of the current transaction which participate in the derivation tree as the reasons for the violation. In traditional databases these objects will be rejected. In our approach we store them as provisioal objects, i.e. as inconsistent objects which will be repaired in the near future.

Each violation is documented within the knowledge base by a special object which references the reasons for the violation. The derivation tree is stored as the justification for the documentation. A simple truth maintenance system manages this justification network. The system do not only check for primary inconsistencies, i.e. constraint violations, but also for secondary inconsistencies, i.e. the satisfaction of an integrity constraint only because of existing provisional objects. This enables us to talk about consequences of the tolerance of inconsistent information within the knowledge base.

5 Related Work

We first compare our approach to work which covers either the modularization of conceptual models or the perspective resolution. In the end we discuss an approach covering both the separation and the resolution of multiple perspectives.

Separation mechanisms have been developed for different purposes in requirements engineering and for modeling environments. The *Requirements Apprentice* [RW91] uses a context mechanism called *Cliché* to represent predefined domain descriptions. Since different domain descriptions may be inconsistent to each other, this separation is necessary. They are organised in a specialization hierarchy and can be used as a starting point in requirements engineering. ARIES [JH91] employs a similar concept called *Folder*. A Folder captures partial domain information and is used for the development of a requirements specification. The engineer creates a new Folder and maybe a relationship to one of the predefined domain descriptions. He then extends this description according to the actual problem domain.

The modularization presented in this paper is compatible with module concepts developed for software architecture languages (as, e.g., [Nag90]). A module is a collection

of information (conceptual models and program statements, resp.) and provides information hiding by encapsulating its content. The communication is supported by import relationships between modules which extend the set of accessible information or resources. To control the communication the contents is divided into a public and a private part. To handle a large number of modules they provide the local definition of modules as a structuring principle. We adapted this for M-Telos and exactly formalized the principles of usability specified for software module hierarchies for deductive databases.

The Cyc Project at MCC is concerned with the development of a knowledge base containing conceptual descriptions of most if not all areas of the real world. To separate the huge amount of information they use a mechanism called *context* [Guh90]. They offer similar properties to the module concept we developed with M-Telos. They can be organised in a specialization hierarchy which makes the whole contents of the general context accessible in the more specific context. This is comparable to our inclusion hierarchy of modules. A most general context called *BaseKB* contains all other contexts as specializations. In contrast to our module concept, there exists a most specific context called *BrowsingCntxt* where all informations of the knowledge base are accessible. It is used to inspect the contents of the knowledge base and is a special case of our resolution module.

Existing perspective resolution approaches are limited to a fixed set of analysis goals and in many cases concentrate on syntactical relationships. Leite and Freeman employ in [LF91] a purely syntatctic perspective comparison. The contents of perspectives are represented by a set of production rules. They compare the rule bases of two perspectives by identifying the most probable rule pairs as well as the rules with no pairs. On basis of the evaluated mapping they detect wrong, missing and inconsistent information. They do not take information about the domain into account. The analysis rules are predefined in the Static Analyzer and cannot be customized by the user.

The viewpoint analysis of Kotonya and Sommerville [KS96] comprises two stages: the correctness and completeness of the viewpoint documentation and the conflict analysis. The completeness of a viewpoint documentation is checked according to the predefined viewpoint structure. This structure defines the required components and attributes of a viewpoint. A problem-oriented definition of the structure by the user is not supported. The conflict analysis is performed by the requirements engineer with the help of the provided toolset.

The ViewPoint approach [NKF94] is a framework for distributed software engineering, in which multiple perspectives are maintained separately as distributable objects called ViewPoints. The necessity to include communication features similar to our module concept is mentioned in [NKF94] and comprises model transfer between ViewPoints and information hiding using interfaces. The integration of ViewPoints is defined by so-called inter-ViewPoint rules. The rules are defined by a method engineer for ViewPoint templates, i.e. for ViewPoint types in which only the notation and the work plan have been defined. The relationships therefore perform always only an integration with respect to the notations; an integration based on the domain that is analysed is not supported.

All rules are bilateral, i.e. they formulate a relationship between a source ViewPoint and a destination ViewPoint. A relationship between multiple ViewPoints must be broken down to bilateral rules. A rule can only be invoked from the source ViewPoint. To allow

both ViewPoints to invoke their relationship it has to be duplicated and refomulated for the destination ViewPoint. The analysis goals used in our approach do not make any assumptions about the perpspectives where they have to be checked. When defining an analysis goal the participants do not need to think about the involved notations and in which direction it should be evaluated. In addition they are not limited to bilateral relationships. At definition time the participants define the goals or questions of the current analysis project and don't have to care how the domain is covered by different notations.

6 Conclusions and Further Work

For some years software specification and design methods have been formalized by a transformation to well-understood formalisms like logic, graph grammars or algebraic specifications to enable a computer-based analysis. It is characteristic of these approaches to assign the methods a fixed semantics the user must accept when using such a system. Beyond that it is assumed that the various partial conceptual models form views on a consistent entire model.

In some other parts of practice just the opposite trend can be observed. Informal teamwork methods leave the details of notations to a great extent to the user and consciously employ conflicts and inconsistencies as an analysis tool, instead of avoiding them. These methods (examples are JAD, SSM and PFR) enjoy increasing popularity exactly because they give negotiation and mutual learning priority over a fixed axiomatization or restriction by reference models. To enhance analysis quality and efficiency formalization and computer support is also desireable for these methods, but they must offer features different to the approaches mentioned above.

In this paper we elaborated these requirements by (i) presenting a comparison of different analysis support methodologies and (ii) by presenting industrial case studies in the business analysis area. On this basis we developed a comprehensive solution for a computer-based support of team- and goal-oriented analysis methods. We extended the formal conceptual modeling language Telos by a separation mechanism called modules, which enables the representation of multiple, conflicting perspectives. We showed that a simple axiomatization of the extended language M-Telos exists, which allows for a realization by well-understood deductive database technology.

We developed the model-based perspective resolution where the knowledge about the structure of the domain and the analysis goals are specified in a meta meta model. The use of M-Telos as representation formalism keeps even the meta meta model customizable. By declaring the notations as partial views on this model we define a connection between the semantic domain description and the syntactic perspective schemata. We used this connection for goal-oriented inconsistency management by the transformation of domain-oriented analysis goals into notation-based integrity constraints. Since in many cases the simple evaluation of cross-perspective relationships is not enough we developed a technique for continuous maintenance of inconsistency information based on deductive database technology.

Our approach is completely implemented in ConceptBase [JGJ+95], a deductive object base manager which uses M-Telos as object model. In cooperation with the German consulting house and software firm USU we applied our approach in several case studies to business process engineering with the PFR analysis method [NZ95]. The indirect support of the formalization and the computer-based support increased the efficiency of the cross-perspective analysis and the quality of analysis results. The system is currently in use by USU in industrial requirements engineering projects.

The four components of our approach can also be used in stand-alone mode together with existing modeling environments or viewpoint mechanisms. The simple axiomatization of the module concept enables its adaption to existing, even non-Telos repositories (as, e.g., the Microsoft Repository [BHS+97]) to represent multiple development perspectives. The combination of the goal-oriented inconsistency management concept with notation-centered CASE tools lead to a more guided modeling process with customizable, domain-oriented integrity constraints. This also works for distributed environments like the ViewPoints approach. Since the constraints are checked locally as before, the central goal definition implies no decrease of system performance.

Data warehousing [Inm96] is concerned with the extraction, integration, aggregation and customization of distributed, heterogenuous operational data. Building, using and managing a data warehouse requires the features we developed in this paper. Data come from multiple sources and may be inconsistent with each other, thus a separation mechanism is needed. To be able to interpret the data right, the existence of conflicts must be monitored and the infected data must be marked.

Acknowledgment: We would like to thank Manfred A. Jeusfeld and Martin Staudt for their contributions to this work. Thanks are also due to our students Lutz Bauer, Farshad Lashgari, Thomas List, Rene Soiron and Christoph Quix for implementing the system.

References

[Abe95] P. Abel. Description of the USU-PFR analysis method. Technical report, USU GmbH, Möglingen, 1995.

[ACM95] S. Abitebuol, S. Cluet, and T. Milo. A database interface for file updates. In M.J. Carey and D.A. Schneider, editors, *Proc. of the 1995 ACM SIGMOD Intl. Conf. on Management of Data*, pages 386–397, May 1995.

[Aug91] J.H. August. *Joint Application Design: The Group Session Approach to System Design.* Yourdon Press, Englewood Cliffs, 1991.

[Bal91] R. Balzer. Tolerating inconsistency. In *Proc. of the 13th Intl. Conf. on Software Engineering (ICSE-13)*, pages 158–165, Austin, Texas, 1991.

[BFG+93] M. Broy, C. Facchi, F. Grosu, R. Hettler, H. Hu"smann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stolen. The requirement and design specification language spectrum: An informal introduction. Technical Report TUM-I9311+I9312, Technische Universität München, 1993.

[BHS+97] P.A. Bernstein, K. Harry, P. Sanders, D. Shutt, and J. Zander. The microsoft repository. In *Proc. of the 23rd Intl. Conf. on Very Large Data Bases (VLDB)*, pages 3–12, Athens, Greece, August 1997.

[Che89] P.B. Checkland. Soft systems methodology. In J. Rosenhead, editor, *Rational Analysis for a Problematic World*, pages 71–100. John Wiley & Sons, Chichester, 1989.

[Cra94] A. Crawford. *Advancing Business Concepts in a JAD Workshop Setting.* Prentice-Hall, Englewood Cliffs, NJ, 1994.

[DCC92] E. Downs, P. Clare, and I. Coe. *Structured Systems Analysis and Design Method.* Prentice-Hall, 2 edition, 1992.

[Eas96] S.M. Easterbrook. Learning from inconsistency. In *Proc. of the 8th Intl. Workshop on Software Specification and Design*, Schloss Velen, Germany, March 1996.

[EN94] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, 1994.

[FF91] M.S. Feather and S. Fickas. Coping with requirements freedoms. In R. Balzer and J. Mylopoulos, editors, *Proc. of the Intl. Workshop on the Development of Intelligent Information Systems*, pages 42–46, Niagara-on-the-Lake, Ontario, Canada, April 1991.

[FS97] M. Fowler and K. Scott. *UML Destilled: Applying the Standard Object Modeling Language.* Addison-Wesley, 1997.

[GJ87] C.F. Gibson and B.B. Jackson. *The Information Imperative: Managing the Impact of Information Technology on Business and People.* Lexington Books D.C. Heath, Lexington, Mass., 1987.

[Gre84] S. Greenspan. *Requirements Modeling: A Knowledge Representation Approach to Requirements Definition.* PhD thesis, Department of Computer Science, University of Toronto, 1984.

[Guh90] R.V. Guha. Micro-theories and contexts in cyc part i: Basic issues. Technical Report MCC Technical Report No. ACT-CYC-129-90, MCC Microelectronics and Computer Technology Corporation, June 1990.

[Hus94] H. Hussmann. Formal foundations of ssadm: An approach integrating the formal and pragmatic world of requirement engineering. Habilitationsschrift, Technische Universität München, June 1994.

[IDS96] IDS Prof. Scheer GmbH, Saarbrücken. *ARIS-Toolset Manual V3.1*, 1996.

[Inm96] W.H. Inmon. *Building the Data Warehouse.* QED Publishing Group, 1996.

[Jeu92] M.A. Jeusfeld. *Update Control in Deductive Object Bases.* PhD thesis, University of Passau (in German), 1992.

[JGJ+95] M. Jarke, R. Gallersdörfer, M.A. Jeusfeld, M. Staudt, and S. Eherer. ConceptBase - a deductive object base for meta data management. *Journal of Intelligent Information Systems, Special Issue on Deductive and Object-Oriented Databases*, 4(2):167–192, March 1995.

[JH91] W.L. Johnson and D.R. Harris. Sharing and reuse of requirements knowledge. In *Proc. of the 6th Annual KBSE Conference, Syracuse*, September 1991.

[KS96] G. Kotonya and I. Sommerville. Requirements engineering with viewpoints. *Software Engineering Journal, Special Issue on Viewpoints in Requirements Engineering*, 11(1):5–18, January 1996.

[LF91] J.C.S.P. Leite and P.A. Freeman. Requirements validation through viewpoint resolution. *IEEE Transactions on Software Engineering*, 17(12):1253–1269, December 1991.

[MBJK90] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems*, 8(4):325–362, October 1990.

[Min87] J. Minker, editor. *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann Publishers, Inc., 1987.

[Nag90] M. Nagl. *Programming-In-The-Large.* Springer-Verlag, 1990.

[Nis96] H.W. Nissen. *Separation and Resolution of Multiple Perspectives in Conceptual Modeling.* PhD thesis, RWTH Aachen, Germany, (in German), 1996.

[NJ97] H.W. Nissen and M. Jarke. Goal-oriented inconsistency management in customizable modeling environments. Technical Report 97-12, RWTH Aachen, Aachener Informatik-Berichte, 1997.

[NJJ+96] H.W. Nissen, M.A. Jeusfeld, M. Jarke, G.V. Zemanek, and H. Huber. Managing multiple requirements perspectives with metamodels. *IEEE Software*, pages 37–47, March 1996.

[NKF94] B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, October 1994.

[NZ95] H.W. Nissen and G.V. Zemanek. Knowledge representation concepts supporting business process analysis. In *Proc. of Reasoning About Structured Objects: Knowledge Representation meets Databases (KRDB-95)*, September 1995.

[Poh94] K. Pohl. The three dimensions of requirements engineering: A framework and its application. *Information Systems*, 19(3), 1994.

[RW91] H.B. Reubenstein and R.C. Waters. The requirements apprentice: Automated assistance for requirements acquisition. *IEEE Transactions on Software Engineering*, 17(3):226–240, March 1991.

[SAP97] SAP. Business blueprint for success. Technical Report SAp Info D&T No. 53, SAP AG, March 1997.

[Sta96] M. Staudt. *View Management in Client-Server Systems.* PhD thesis, RWTH Aachen, (in German), 1996.

[You89] E. Yourdon. *Modern Structured Analysis.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

A Complete List of M-Telos Axioms

Predefined objects:

(A-1) $P(\#Obj, \#Obj, Object, \#Obj)$

(A-2) $P(\#Indiv, \#Indiv, Individual, \#Indiv)$

(A-3) $P(\#attr, \#Obj, attribute, \#Obj)$

(A-4) $P(\#Inst, \#Obj, InstanceOf, \#Obj)$

(A-5) $P(\#Isa, \#Obj, IsA, \#Obj)$

Specification of object *Module*:

(A-6) $P(\#Mod, \#Mod, Module, \#Mod)$

(A-7) $P(\#cont, \#Mod, contains, \#Obj)$

(A-8) $P(\#exp, \#Mod, exports, \#Obj)$

(A-9) $P(\#imp, \#Mod, imports\_from, \#Mod)$

(A-10) $P(\#coord, \#Mod, coordinates, \#Mod)$

The contents of module *System*:

(A-11.1) $P(\#Sys, \#Sys, System, \#Sys)$

(A-11.2) $P(\#Sysin, \#Sys, in, \#Mod)$

(A-11.3) $P(\#SysC1, \#Sys, c1, \#Obj)$

(A-11.4) $P(\#SysI1, \#SysC1, in, \#cont)$

(A-11.5) $P(\#SysC2, \#Sys, c2, \#Indiv)$

(A-11.6) $P(\#SysI2, \#SysC2, in, \#cont)$

(A-11.7) $P(\#SysC3, \#Sys, c3, \#attr)$

(A-11.8) $P(\#SysI3, \#SysC3, in, \#cont)$

(A-11.9) $P(\#SysC4, \#Sys, c4, \#Inst)$

(A-11.10) $P(\#SysI4, \#SysC4, in, \#cont)$

(A-11.11) $P(\#SysC5, \#Sys, c5, \#Isa)$

(A-11.12) $P(\#SysI5, \#SysC5, in, \#cont)$

(A-11.13) $P(\#SysC6, \#Sys, c6, \#Mod)$

(A-11.14) $P(\#SysI6, \#SysC6, in, \#cont)$

(A-11.15) $P(\#SysC7, \#Sys, c7, \#cont)$

(A-11.16) $P(\#SysI7, \#SysC7, in, \#cont)$

(A-11.17) $P(\#SysC8, \#Sys, c8, \#exp)$

(A-11.18) $P(\#SysI8, \#SysC8, in, \#cont)$

(A-11.19) $P(\#SysC9, \#Sys, c9, \#imp)$

(A-11.20) $P(\#SysI9, \#SysC9, in, \#cont)$

(A-11.21) $P(\#SysC10, \#Sys, c10, \#coord)$

(A-11.22) $P(\#SysI10, \#SysC10, in, \#cont)$

(A-11.23) $P(\#SysC11, \#Sys, c11, \#Sys)$

(A-11.24) $P(\#SysI11, \#SysC11, in, \#cont)$

(A-11.25) $P(\#SysC12, \#Sys, c12, \#Sysin)$

(A-11.26) $P(\#SysI12, \#SysC12, in, \#cont)$

Object identity:

(A-12) $P(o,x_1,l_1,y_1) \wedge P(o,x_2,l_2,y_2) \Rightarrow (x_1 = x_2) \wedge (l_1 = l_2) \wedge (y_1 = y_2)$

Drived literals:

(A-13) $P(o,i,in,c) \Rightarrow In(i,c)$

(A-14) $P(o,x,l,y) \wedge P(p,c,m,d) \wedge In(o,p) \Rightarrow A(x,m,y)$

Uniqueness of module names:

(A-15) $P(o_1,o_1,m,o_1) \wedge P(o_2,o_2,m,o_2) \wedge In(o_1,\#Mod) \wedge In(o_2,\#Mod) \Rightarrow (o_1 = o_2)$

Uniqueness of individual names:

(A-16) $In(M,\#Mod) \wedge P(o_1,o_1,l,o_1) \wedge P(o_2,o_2,l,o_2) \wedge A(M,contains,o_1) \wedge$
$A(M,contains,o_2) \Rightarrow (o_1 = o_2)$

Uniqueness of attribute names:

(A-17) $P(o_1,s,l,d_1) \wedge P(o_2,s,l,d_2) \wedge In(M,\#Mod) \wedge A(M,contains,o_1) \wedge$
$A(M,contains,o_2) \Rightarrow (o_1 = o_2) \vee (l = in) \vee (l = isa)$

Objects belong to exactly one module:

(A-18) $P(o,x,l,y) \Rightarrow \exists M\ In(M,\#Mod) \wedge A(M,contains,o)$

(A-19) $In(M,\#Mod) \wedge In(N,\#Mod) \wedge P(o,x,l,y) \wedge A(M,contains,o) \wedge$
$A(N,contains,o) \Rightarrow (M = N)$

Derived contains relations for a modules attributes:

(A-20) $In(M,\#Mod) \wedge P(o,M,l,y) \wedge (In(o,\#cont) \vee In(o,\#imp) \vee$
$In(o,\#exp) \vee In(o,\#coord)) \Rightarrow A(M,contains,o)$

(A-21) $In(M,\#Mod) \wedge P(o,M,l,y) \wedge (P(x,o,in,\#cont) \vee P(x,o,in,\#imp) \vee$
$P(x,o,in,\#exp) \vee P(x,o,in,\#coord)) \Rightarrow A(M,contains,x)$

Definition of predicate P^{Mod}:

(A-22) $In(M,\#Mod) \wedge P(o,x,l,y) \wedge A(M,contains,o) \Rightarrow P^{Mod}(M,o,x,l,y)$

(A-23) $In(M,\#Mod) \wedge In^{Mod}(M,N,\#Mod) \wedge P^{Mod}(N,o,x,l,y) \wedge$
$A^{Mod}(M,M,imports\_from,N) \wedge A^{Mod}(N,N,exports,o) \Rightarrow P^{Mod}(M,o,x,l,y)$

(A-24) $In(M,\#Mod) \wedge In^{Mod}(M,N,\#Mod) \wedge P^{Mod}(N,o,x,l,y) \wedge$
$A^{Mod}(M,M,coordinates,N) \Rightarrow P^{Mod}(M,o,x,l,y)$

(A-25) $In^{Mod}(N,M,\#Mod) \wedge In(N,\#Mod) \wedge A^{Mod}(N,N,contains,M) \wedge$
$P(N,o,x,l,y) \Rightarrow P^{Mod}(M,o,x,l,y)$

Definition of predicates $In^{Mod}, Isa^{Mod}, A^{Mod}$:

(A-26) $P^{Mod}(M,o,i,in,c) \Rightarrow In^{Mod}(M,i,c)$

(A-27) $P^{Mod}(M,o,x,l,y) \wedge P^{Mod}(M,p,c,m,d) \wedge In^{Mod}(M,o,p) \Rightarrow A^{Mod}(M,x,m,y)$

(A-28) $P^{Mod}(M,0,c,isa,d) \Rightarrow Isa^{Mod}(M,c,d)$

Membership to the builtin classes:

(A-29) $P^{Mod}(M,o,s,l,d) \Rightarrow In^{Mod}(M,o,\#Obj)$

(A-30) $In^{Mod}(M,o,\#Obj) \Rightarrow \exists\, s,l,d\; P^{Mod}(M,o,s,l,d)$

(A-31) $P^{Mod}(M,o,o,l,o) \Rightarrow In^{Mod}(M,o,\#Indiv)$

(A-32) $In^{Mod}(M,o,\#Indiv) \Rightarrow \exists\, l\; P^{Mod}(M,o,o,l,o)$

(A-33) $P^{Mod}(M,o,i,in,c) \Rightarrow In^{Mod}(M,o,\#Inst)$

(A-34) $In^{Mod}(M,o,\#Inst) \Rightarrow \exists\, i,c\; P^{Mod}(M,o,i,in,c)$

(A-35) $P^{Mod}(M,o,c,isa,d) \Rightarrow In^{Mod}(M,o,\#Spec)$

(A-36) $In^{Mod}(M,o,\#Spec) \Rightarrow \exists\, c,d\; P^{Mod}(M,o,c,isa,d)$

(A-37) $P^{Mod}(M,o,s,l,d) \wedge (o \neq s) \wedge (o \neq d) \wedge (l \neq in) \wedge (l \neq isa) \Rightarrow In^{Mod}(M,o,\#Attr)$

(A-38) $In^{Mod}(M,o,\#Attr) \Rightarrow \exists\, s,l,d\; P^{Mod}(M,o,s,l,d) \wedge (o \neq s) \wedge (o \neq d) \wedge$
$(l \neq in) \wedge (l \neq isa)$

Any object falls into one of the four categories:

(A-39) $In^{Mod}(M,o,\#Obj) \Rightarrow In^{Mod}(M,o,\#Indiv) \vee In^{Mod}(M,o,\#Inst) \vee$
$In^{Mod}(M,o,\#Spec) \vee In^{Mod}(M,o,\#Attr)$

The isa relation is a partial order:

(A-40) $In^{Mod}(M,c,\#Obj) \Rightarrow Isa^{Mod}(M,c,c)$

(A-41) $Isa^{Mod}(M,c,d) \wedge Isa^{Mod}(M,d,e) \Rightarrow Isa^{Mod}(M,c,e)$

(A-42) $Isa^{Mod}(M,c,d) \wedge Isa^{Mod}(M,d,c) \Rightarrow (c = d)$

Inheritance of class membership:

(A-43) $In^{Mod}(M,i,c) \wedge P^{Mod}(M,o,c,isa,d) \Rightarrow In^{Mod}(M,i,d)$

Attributes are typed by the atribute classes:

(A-44) $P^{Mod}(M,o,s,l,d) \wedge In^{Mod}(M,o,p) \Rightarrow \exists\, c,m,k\; P^{Mod}(M,p,c,m,k) \wedge$
$In^{Mod}(M,s,c) \wedge In^{Mod}(M,d,k)$

Consistency of a specialization relationship:

(A-45) $Isa^{Mod}(M,o_1,o_2) \wedge P^{Mod}(M,o_1,c,l_1,e) \wedge P^{Mod}(M,o_2,d,l_2,f)$
$\Rightarrow Isa^{Mod}(M,c,d) \wedge Isa^{Mod}(M,e,f)$

Attribute refinement:

(A-46) $Isa^{Mod}(M,c,d) \wedge P^{Mod}(M,o_1,c,l,e) \wedge P^{Mod}(M,o_2,d,l,f)$
$\Rightarrow Isa^{Mod}(M,e,f) \wedge Isa^{Mod}(M,o_1,o_2)$

Multiple classification:

(A-47) $In^{Mod}(M,i,c) \wedge In^{Mod}(M,i,d) P^{Mod}(M,o_1,c,l,f) \wedge P^{Mod}(M,o_2,d,l,g)$
$\Rightarrow \exists\, e,h,o_3\; In^{Mod}(M,i,e) \wedge P^{Mod}(M,o_3,e,l,h) \wedge$
$Isa^{Mod}(M,e,c) \wedge Isa^{Mod}(M,e,d)$

The export part satisfies referential integrity:

(A-48) $In(M,\#Mod) \wedge P(o,x,l,y) \wedge A^{Mod}(M,M,exports,o) \Rightarrow P^{Mod}(M,o,x,l,y)$

(A-49) $In(M, \#Mod) \wedge A^{Mod}(M, M, exports, o) \wedge P^{Mod}(M, o, x, l, y)$
$\Rightarrow A^{Mod}(M, M, exports, x) \wedge A^{Mod}(M, M, exports, y)$

Constraints of the *coordinates* relationship:

(A-50) $In(M, \#Mod) \wedge In(N, \#Mod) \wedge In(P, \#Mod) \wedge A(M, coordinates, N) \wedge$
$A(N, coordinates, P) \Rightarrow A(M, coordinates, P)$

(A-51) $In(M, \#Mod) \wedge In(N, \#Mod) \wedge A(M, coordinates, N) \wedge$
$A(N, coordinates, M) \Rightarrow (M = N)$

Formal Models and Prototyping*

Luqi
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Abstract

Rapid prototyping is a promising approach for formulating accurate software requirements, particularly for complex systems with hard real-time constraints. Computer aid is needed for realizing the potential benefits of this approach in practice, because the problems associated with software evolution are greatly amplified in the context of iterative prototyping and exploratory design.

Our computer-aided prototyping system CAPS provides automated support for many aspects of requirements analysis and software prototyping, including: (1) maintaining logical dependencies between assumptions about needs of different groups, software requirements, and design decisions, (2) managing design history, alternatives and dependencies, (3) planning, assigning and scheduling job assignments for teams of designers in the presence of uncertainty, (4) checking and propagating design constraints, (5) maintaining consistency between graphical and text views of a design, (6) constructing real-time schedules, (7) generating control code, and (8) retrieving and instantiating reusable software components.

The principles and methods that make this possible and the practical application of the system are explained via examples.

1 Introduction

The software industry remains far short of the mature engineering disciple needed to meet the demands of our information age society. Symptoms of the problem are large sums spent on cancelled software projects [38], costly delays [19], and software reliability problems [13].

Lack of formalization of rapidly emerging application areas makes software engineering more difficult than other engineering disciplines. Requirements for complex systems are nearly always problematic initially and evolve throughout the life of the systems. Requirements and specification problems have been found to be the dominant cause of faults in the Voyager/Galileo software [34], and we believe this applies to most large and complex systems.

*This research was supported in part by the National Science Foundation under grant number CCR-9058453 and by the Army Research Office under grant number 30989-MA.

Evolutionary prototyping can alleviate this problem by providing an efficient approach to formulating accurate software requirements [27]. Simple models reflecting the main issues associated with the proposed system are constructed and demonstrated, and then reformulated to better match customer concerns, based on specific criticisms and the issues they elicit. This process aids understanding because independent issues are separated and treated in isolation as much as possible, via communication based on the simplest models possible. The models are refined only as needed to resolve open issues, and the issues arising at one level of detail are resolved as much as possible before considering the next level of detail, or the next aspect of the system. This helps to focus the attention of the customers, designers, and analysts because only a few selected aspects of the system are changing at any point in the process.

Automation is necessary to enable the rapid, economical and effective change needed for evolutionary prototyping. Our hypothesis has been that increasing the degree of automation for system development and evolutionary prototyping should improve the quality of the systems produced. A sound basis for the engineering automation is needed to realize evolutionary prototyping for large and complex systems, which typically have real-time constraints. We have explored formal models of various aspects of software development and evolution to achieve reliable and quantifiable automation of subtasks. Formal models have enabled analysis and assessment of the accuracy and efficiency of proposed algorithms and heuristics.

It has been necessary to interleave this theoretical work with experimental validation and adjustment of the models to better fit practical reality. This has been necessary because software development and evolution are extremely complex problem domains, and engineering automation systems have correspondingly complex requirements that strongly manifest all of the difficulties identified above. Thus we have applied the evolutionary prototyping approach to the development of techniques and software for supporting the evolutionary prototyping approach itself. We have found this strategy successful for developing accurate models, effective automation and decision support methods for evolution of software and system requirements. This paper summarizes our experiences and presents some of our recent progress on carrying out the plan outlined above.

The rest of the paper is organized as follows. Section 2 describes our strategies for achieving automation support for evolutionary prototyping and summarizes progress to date. Section 3 discusses a formal model of software evolution and explores some automated processes that can be supported by the model. Section 4 illustrates our ideas with an example. Section 5 contains conclusions.

2 Strategy for Automation Support

The main components of our strategy are developing languages and methods based on formal models of selected aspects of the problem. In each case we sought the simplest models adequate for achieving our purposes, and based the languages and methods on these models. We started with the simplest possible models and refined them only as needed, based on experimental application of the models to assess their adequacy. Our guiding principle was to avoid model features unless we has a convincing practical scenario that required those features. Consequently we were always searching for simplifications

and reformulated models whenever we found a way to eliminate a model concept. This was done because we wanted the resulting methods and tools to be easy to use and learn. We expected simpler models to speed up the processes of analysis and design by reducing the number of mandatory choices. This is particularly appropriate in the context of prototyping, where it is important to get the major decisions correct rapidly, without spending effort on fine-tuning. Our experience has confirmed this hypothesis. We have also found that removing concepts from the models and the attention of the designer can introduce stringent requirements for design automation capabilities. Removed design attributes must be derived automatically, accurately, and in a way that provides good designs.

The first area to be modeled was the behavior of real-time systems, because the prototyping approach requires the ability to demonstrate proposed system behavior. The simplest formulation we could find was a refinement of data flow models that incorporates declarative control and timing constraints. The prototype system description language PSDL [25] was developed based on this model. The model was extended to include distributed computation [30] and a formal semantics of the language was developed [22]. The model and language have been found to be adequate for representing a variety of complex systems, including a generic C3I station [29] and a wireless acoustic monitor for preventing sudden infant death syndrome [36].

Real-time scheduling and software integration are other key issues for rapid realization of complex systems. We developed related models in these two areas, based on the model of system behavior.

Real-time scheduling depends on models of the timing requirements and on models of the capabilities of the target hardware. The behavioral model underlying PSDL contains a model of real-time requirements, which we extracted for this purpose [26]. This model was used to develop our initial scheduling methods, and it proved adequate. The initial hardware model was empty, which was adequate for scheduling with respect to fixed, single processor architectures. We realized that scheduling depended on hardware models when we started addressing scheduling methods for more general hardware configurations. We developed a series of more sophisticated hardware models [30], and found that these together with the original model of real-time requirements were adequate for supporting scheduling methods for multi-processor and distributed target hardware configurations [7, 32]. Ongoing work is exploring models and methods that can schedule larger distributed real-time computations within practical resource limits.

Software integration is the process of ensuring that all the parts of a software system work together to achieve their intended purpose. Software integration depends on models of interactions between subsystems and control constraints, including those derived from timing requirements and the schedules used to realize them. We addressed software integration by developing software architectures and methods for architecture-based program generation. Automated program generation is necessary in our context because we had to support rapid, low cost change, and small changes to timing requirements can affect large portions of the code.

The software architecture for prototypes embodies a general structure for realizing interactions and real-time schedules for systems that have a mix of time-critical and non time-critical computations. The structure used to automatically realize connections between subsystems was derived from the system interaction part of the behavioral model

underlying PSDL. The structure that realizes the schedules uses a high-priority thread for computations with hard deadlines and a low-priority thread for computations without deadlines.

The software architecture was implicitly defined by a program generator driven by a description of desired system behavior expressed in PSDL and a real-time schedule constructed by a scheduling algorithm. This program generator was itself generated using an attribute grammar processor. This approach works but is not particularly elegant or easy to adapt to other problems.

Our initial capability for generating executable prototypes from simple and quickly constructible models of the problem domain enabled experimental validation of the conjecture that prototyping and demonstrations of systems behavior were valuable aids to requirements determination. The initial experiments supported the validity of this conjecture, which motivated us to put more effort into software reuse and evolution.

Software reuse is a critical part of prototyping for real-time systems because efficiency is of the essence in the time-critical parts of these systems. The highest levels of efficiency can only be achieved by intensive engineering and refinement of sophisticated algorithms and data structures, which usually takes large amounts of time and effort, and produces designs that depend on intricate chains of reasoning. The easiest way to take advantage of such components in a process that must be cheap and rapid is to use a previously constructed library of standard and well-optimized components. Thus we explored formal models of how such libraries could be organized and searched to quickly find the most appropriate components for each particular context [24]. Search methods must trade off precision (retrieving only relevant components) against recall (finding components if they are relevant). We have developed a software component search method that can simultaneously achieve high levels of precision and recall, based on algebraic queries representing symbolic test cases.

Software evolution is a critical aspect of prototyping [27]. In the early stages of requirements formulation the purposes of the proposed system are highly uncertain and major changes are expected. Planning, version control, team coordination, and project management are key issues in this context. Another important issue is how to repeatedly and rapidly change a design without having it degenerate into an unstructured maze that cannot be quickly understood and modified. The next section summarizes our progress on software evolution.

3 Software Evolution

Our initial step towards formalizing software evolution in the large was a graph model of the evolution history [28]. This work led to the insights that the essence of project history lies in dependencies among versions of project documents and the activities that produce them, that the formal structures of project history and project plans are essentially the same, and that integrated modeling and support for software configuration management and project management enables higher automation levels for both [1]. More recent work suggested that hypergraphs may be useful [33], and that integration with personnel models and rationale models enables decision support for the problematic early stages of critique analysis and change planning [8].

To achieve simplicity, we seek to model the products and processes involved in software evolution using a minimal set of general object types, and introduce specialized subclasses only when necessary for accurate modeling. The current version of the model has only three main types: component, step, and person.

The type component represents any kind of versioned software-related object, including critiques, issues, requirements, designs, programs, manuals, test cases, plans, etc. These are the information products produced by software evolution processes.

The type step represents instances of any kind of scheduled software evolution activity, such as analysis, design, implementation, testing, inspection, demonstration, etc. Steps are activities that are usually carried out by people, and may be partially or completely automated. When viewed in the context of evolution history, steps represent dependencies among components. Steps that are not yet completed represent plans. Steps are a subclass of component because they can have versions, to provide a record of how the project plans evolved.

The type person represents the people involved in the software evolution activity, including the stakeholders of the software system, software analysts, designers, project managers, testers, software librarians, system administrators, etc. We need to represent the people involved to be able to trace requirements back to the original raw data, and to link it to the roles the authors of critiques play in the organizational structure. This is a part of the rationale of the system that helps to identify viewpoints and analyze tradeoffs between conflicting requirements. The people in the development team must be modeled because of concerns related to project scheduling and authorization to access project information. Person is also a subclass of component, and therefore versioned, to provide a record of how the roles and qualifications of the people involved in the project change with time.

We have recently developed an improved model of system evolution that better accounts for hierarchical structures of components and steps. The associated refinement concept is useful for helping developers and planners to cope with the complexity of large projects. This model is summarized as follows.

An **evolution record** is a labeled acyclic directed hypergraph $[N, E, I, O, C, S]$ where

1. N is a set of **nodes**, representing unique identifiers for components,
2. E is a set of **edges**, representing unique identifiers for steps,
3. $I : E \rightarrow 2^N$ is a function giving the set of **inputs** of each edge,
4. $O : E \rightarrow 2^N$ is a function giving the set of **outputs** of each edge, such that $O(e) \cap O(e') \neq \emptyset$ implies $e = e'$,
5. $C : N \rightarrow \textit{component}$ is a function giving the **component** associated with each node, and
6. $S : E \rightarrow \textit{step}$ is a function giving the **step** associated with each edge.

The hypergraph must be acyclic because its edges represent input/output dependencies for the processes that create components. These dependencies induce precedence constraints for the project schedule, because an activity cannot start until all of its input

components are available. The restriction on the outputs says that each component is produced by a unique step. This establishes clear lines of responsibility and produces a record of authorship when each step completes.

Let H denote the set of evolution records.

A **hierarchical evolution record** is an acyclic directed graph $[n, e]$ with label maps h, r and decomposition maps d_n, d_e where

1. n is a set of **nodes** representing unique identifiers for evolution records.
2. e is a set of **edges** representing unique identifiers for evolution record refinements,
3. $h : n \rightarrow H$ is a function giving the **evolution record** associated with each node, such that $(n_1, n_2) \in e$ implies $h(n_1)$ is a subhypergraph of $h(n_2)$. This means that $h(n_1).N \subseteq h(n_2).N$, $h(n_1).E \subseteq h(n_2).E$, $h(n_1).I \subseteq h(n_2).I$, $h(n_1).O \subseteq h(n_2).O$, $h(n_1).C \subseteq h(n_2).C$, and $h(n_1).S \subseteq h(n_2).S$.
4. $r : e \rightarrow \mathit{step}$ is a function giving the **step** that is refined by each edge,
5. $d_n : N \rightarrow 2^N$ is a function giving the set of **subcomponent nodes** of each component node appearing in the evolution record $h(n_i)$ for any node $n_i \in n$, where $N = \bigcup_{n_i \in n} h(n_i).N$.
6. $d_e : E \rightarrow 2^E$ is a function giving the set of **substep edges** of each step edge appearing in the evolution record $h(n_i)$ for any node $n_i \in n$, where $E = \bigcup_{n_i \in n} h(n_i).E$.
7. The graph has a single root (a node with no incoming edges) and a single leaf (a node with no outgoing edges).
8. Any two paths p_1 and p_2 from the root node with the same step label set $\{r(e) | e \in p_1\} = \{r(e) | e \in p_2\}$ end in the same node.
9. If $(n_1, n_2) \in e$, then there is an $E_1 \in h(n_1).E$ with $S(E_1) = r(e)$, $\emptyset \neq d_e(E_1) \subseteq h(n_2).E$, and for each $E_2 \in d_e(E_1)$, $I(E_2) \subseteq \bigcup_{N_1 \in I(E_1)} d_n(N_1) \subseteq h(n_2).N$ and $O(E_2) \subseteq \bigcup_{N_1 \in O(E_1)} d_n(N_1) \subseteq h(n_2).N$.

Each node of a hierarchical evolution record represents a view of the evolution history. The root node is the most abstract view, containing only the top level steps and the top level components those steps produce. The leaf node is the most detailed view, which contains the top level steps and components together with all direct and indirect substeps and subcomponents.

A step is refined by adding all of its substeps to the evolution record, along with the input and output components of the substeps. The last condition in the definition says that the step associated with the link between two views must be decomposed into at least one substep in the detailed view, that the inputs and outputs of the substeps must be subcomponents of the inputs and outputs of the superstep, and that the input and output components of the substeps must appear in the detailed view.

The hierarchical evolution record has a large number of nodes, which are not intended to be stored explicitly in an implementation. The model is intended as a framework for navigation through the possible views of the evolution record at different levels of

abstraction. Practical implementations will materialize only those view nodes that are visited.

This model can be used to automatically schedule steps, automatically locate and deliver the proper versions of the input components to the developer assigned to carry out the step, and to automatically check in the new components produced when the step is completed. It can also be used to generate default plans, to maintain the consistency of plans, and to help managers and developers navigate through the plan and document structures of an evolutionary prototyping or development effort.

4 Example

Figure 1 shows an example of a top level evolution record. In this example, the first version of the requirement ($R1$) is used to derive the first version of the prototype ($P1$), which is demonstrated to system stakeholders and elicits the criticism ($C1$). When a step to derive the second version of the requirement ($R2$) from the criticism is proposed, the system automatically proposes a step to create the second version of the prototype ($P2$), because the prototype depends on the requirement and the requirement will be updated. The proposed steps will be scheduled automatically when they are approved by the project management.

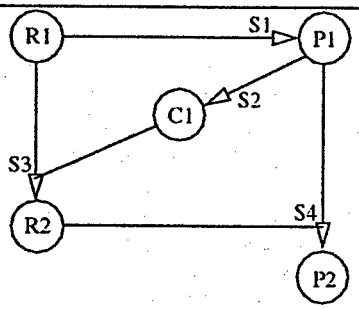


Figure 1: Top Level Evolution Record

Figure 2 shows the refinement of step $S1$ of the top level evolution record shown in Figure 1. Both $S1$ and its substeps $S1.1$ and $S1.2$ are present in the refined evolution record. The top level steps are shown with thicker lines. The component $R1$ is decomposed into the subcomponents $Ra1$ and $Rb1$ because these components are inputs to the substeps, and $P1$ is decomposed into $Pc1$ and $Pd1$ because these are the outputs of the substeps.

Figure 3 shows a further refinement of the evolution record shown in Figure 2 that expands all of the top-level steps. We have left out the top level steps to avoid cluttering the diagram. Note that the subrequirement $Rb1$ is shared by both versions of the requirement R, because it is not affected by the elicited criticism, and that the subsystem $Pd1$ of the prototype that depends only on this subrequirement is also shared by both versions

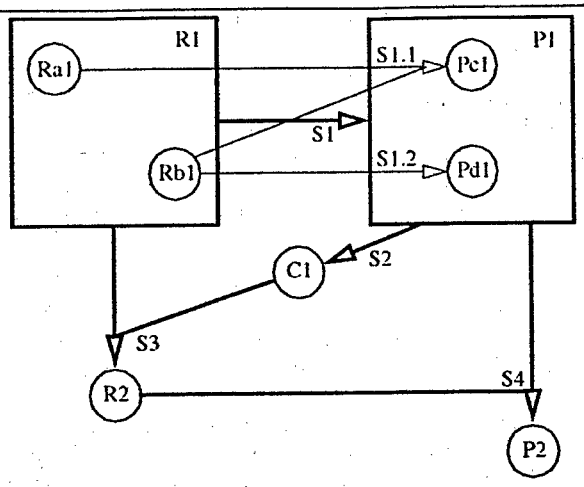


Figure 2: Refinement of Step S1

of the prototype P. Our goal is to provide tools based on this model that will make it easier to discover and manage large scale structures of this variety.

The decomposition mappings for the subcomponents are denoted by geometrical containment in the figures. The decomposition relations for the steps are indicated only via the structure of the step names. Note that the graphical display would get crowded if the decomposition relations were explicitly displayed as hyper-edges, even for this very small example. In realistic situations, there can be many more nodes in the evolution records. We are currently exploring automatic mechanisms for determining and displaying small neighborhoods of these structures that are relevant to particular planning and analysis tasks and are small enough to be understood. Some initial results along these lines can be found in [23].

5 Conclusions

Our previous research has explored formal models of the chronological evolution history [28]. This model has been applied to automate configuration management and a variety of project management functions [1]. The ideas presented in this paper provide a basis for improving these capabilities, particularly in the area of computer aid for understanding the record of the evolution of the system to extract useful information from it. Some recent work on improving the project scheduling algorithms based on these models has enabled scheduling 100,000 tasks in less than a minute [14]. These results suggest that the project scheduling support will scale up to projects of formidable size.

We are currently working on models and notations that support explicit definitions of software architectures for solving given classes of problems independently from the rules

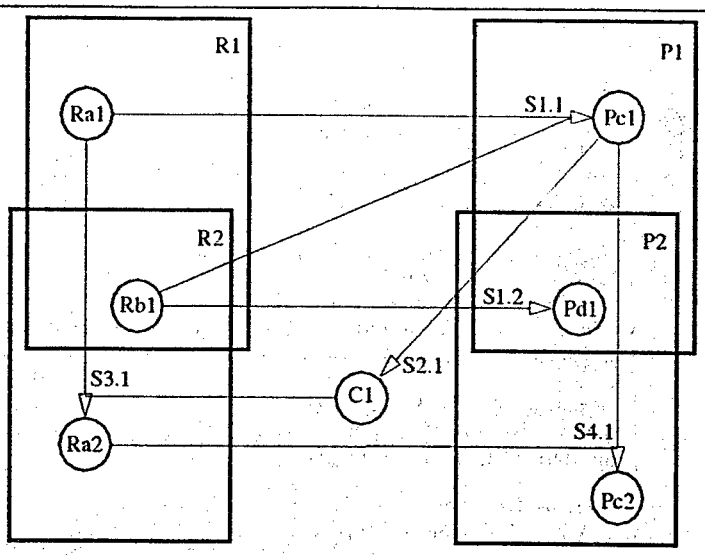


Figure 3: Further Refined Evolution Record

that determine a particular instance of the architecture for solving a given instance of the class of problems. This should make it easier for software architectures and associated program generation capabilities to evolve.

Architecture evolution provides a practical path for quickly obtaining automation capabilities for new problem domains, and to gradually improve those capabilities by adding solution techniques that expand the problem domain and incorporating optimizations for specialized subproblems that improve performance.

Formalizing these aspects of software architectures and developing the corresponding engineering automation methods will eventually enable us to certify that all programs possibly generated from a mature architecture are free from given classes of faults or that they work correctly for all possible inputs. These steps will bring us closer to the point where product-quality software can be economically produced using the same engineering automation technology that enables evolutionary prototyping and helps analysts home in on good requirements models. Our vision is to eliminate the current conflict between rapid development and high software quality.

Our ultimate research goal is to create conceptual models and software tools that allow automatic generation of variations on a software system with human consideration of only the highest-level decisions that must change between one version and the next. Realization of this goal will lead to more flexible software systems and should make prototyping and exploratory design more effective.

References

[1] S. Badr, Luqi, Automation Support for Concurrent Software Engineering. *Proc. of the 6th International Conference Software Engineering and Knowledge Engineering*, Jurmala, Latvia, June 20-23, 1994, 46-53.

[2] F. Bauer et al., *The Munich Project CIP. Volume II: The Program Change System CIP-S*, Lecture Notes in Computer Science 292, Springer 1987.

[3] V. Berzins, On Merging Software Enhancements *Acta Informatica*, Vol. 23 No. 6, Nov 1986, pp. 607-619.

[4] V. Berzins, Luqi, An Introduction to the Specification Language Spec, *IEEE Software*, Vol. 7 No. 2, Mar 1990, pp. 74-84.

[5] V. Berzins, Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada*, Addison-Wesley Publishing Company, 1991, ISBN 0-201-08004-4.

[6] V. Berzins, Software Merge: Models and Methods, *Journal of Systems Integration*, Vol. 1, No. 2, pp. 121-141 Aug 1991.

[7] V. Berzins, Luqi, M. Shing, Real-Time Scheduling for a Prototyping Language, *Journal of Systems Integration*, Vol. 6, No. 1-2, pp. 41-72, 1996.

[8] V. Berzins, O. Ibrahim, Luqi, A Requirements Evolution Model for Computer Aided Prototyping *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, Madrid, Spain, June 17-20, 1997, pp. 38-47.

[9] D. Dampier, Luqi, V. Berzins, Automated Merging of Software Prototypes. *Journal of Systems Integration*, Vol. 4, No. 1, February, 1994, pp. 33-49.

[10] V. Berzins, Software Merge: Semantics of Combining Changes to Programs. *ACM TOPLAS*, Vol. 16, No. 6, Nov. 1994, 1875-1903.

[11] V. Berzins, *Software Merging and Slicing*, IEEE Computer Society Press Tutorial, 1995, ISBN 0-8186-6792-3.

[12] V. Berzins, D. Dampier, Software Merge: Combining Changes to Decompositions. *Journal of Systems Integration*, special issue on CAPS (Vol. 6, No. 1-2, March 1996), pp. 135-150.

[13] M. Dowson, The ARIANE 5 Software Failure, *ACM Software Engineering Notes*, Vol. 22 No. 2, March 1997, p. 84.

[14] J. Evans, Software Project Scheduling Tool, MS Thesis, Computer Science. Naval Postgraduate School, Sep. 1997.

[15] M. Feather, A System for Assisting Program Change. *ACM Transactions on Programming Languages and Systems*, Vol. 4 No. 1, Jan 1982, pp. 1-20.

[16] M. Feather, A Survey and Classification of some Program Change Approaches and Techniques, in *Program Specification and Change (Proceedings of the IFIP TC2/WG 2.1 Working Conference)*, L.G.L.T. Meertens, Ed., North-Holland, 1987, pp. 165-195.

[17] M. Feather, Constructing Specifications by Combining Parallel Elaborations, *IEEE Transactions on Software Engineering*, Vol. 15 No. 2, Feb 1989, pp. 198 208.

[18] S. Fickas, Automating the Transformational Development of Software, *IEEE Transactions on Software Engineering*, Vol. 11 No. 11, Nov 1985, pp. 1268 1277.

[19] W. Gibbs, Software's Chronic Crisis, *Scientific American*, SEP 1994, pp. 86-94.

[20] W. Johnson, M. Feather, Building an Evolution Change Library, *12th International Conference on Software Engineering*, 1990, pp. 238-248.

[21] E. Kant, On the Efficient Synthesis of Efficient Programs, *Artificial Intelligence*, Vol. 20 No. 3, May 1983, pp. 253-36. Also appears in [35], pp. 157-183.

[22] B. Kraemer, Luqi, V. Berzins, Compositional Semantics of a Real-Time Prototyping Language *IEEE Transactions on Software Engineering*, Vol. 19, No. 5, pp. 453-477, May 1993.

[23] D. Lange, Hypermedia Analysis and Navigation of Domains, MS Thesis, Computer Science, Naval Postgraduate School, Sep. 1997.

[24] Luqi, M. Ketabchi, A Computer Aided Prototyping System, *IEEE Software*, Vol. 5 No. 2, Mar 1988, pp. 66-72.

[25] Luqi, V. Berzins, R. Yeh, A Prototyping Language for Real-Time Software, *IEEE Transactions on Software Engineering*, Vol. 14 No. 10, Oct 1988, pp. 1409-1423.

[26] Luqi, Handling Timing Constraints in Rapid Prototyping *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, IEEE Computer Society, Jan. 1989, pp. 417-424.

[27] Luqi, Software Evolution via Rapid Prototyping, *IEEE Computer*, Vol. 22, No. 5, May 1989, pp. 13-25.

[28] Luqi, A Graph Model for Software Evolution, *IEEE Transactions on Software Engineering*, Vol. 16, No. 8, pp. 917-927, Aug. 1990.

[29] Luqi, Computer-Aided Prototyping for a Command-and-Control System Using CAPS, *IEEE Software*, Vol. 9, No. 1, pp. 56-67, Jan. 1992.

[30] Luqi, Real-Time Constraints in a Rapid Prototyping Language, *Journal of Computer Languages*, Vol. 18, No. 2, pp. 77-103, Spring 1993.

[31] Luqi, Specifications in Software Prototyping, *Proc. SEKE 96*, Lake Tahoe, NV, June 10-12, 1996, pp. 189-197.

[32] Luqi, Scheduling Real-Time Software Prototypes, *Proceedings of the 2nd International Symposium on Operations Research and its Applications*, Guilin, China, December 11-13, 1996, pp. 614-623.

[33] Luqi, J. Goguen, Formal Methods: Promises and Problems, *IEEE Software*, Vol. 14, No. 1, Jan. 1997, pp. 73-85.

[34] R. Lutz, Analyzing Software Requirements: Errors in Safety-Critical Embedded Systems, TR 92-27, Iowa State University, AUG 1992.

[35] C. Rich, R. Waters, Eds., *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[36] D. Rusin, Luqi, M. Scanlon, SIDS Wireless Acoustic Monitor (SWAM), *Proc. 21st Int. Conf. on Lung Sounds*, Chester, England, International Lung Sounds Association, Sep. 4-6, 1996.

[37] D. Smith, G. Kotik, S. Westfold, Research on Knowledge-Based Software Environments at Kestrel Institute, *IEEE Transactions on Software Engineering*, Vol. 11 No. 11, Nov 1985, pp. 1278-1295.

[38] Chaos, Technical Report, The Standish Group, Dennis, MA, 1995, http://www.standishgroup.com/chaos.html.

[39] W. Swartout, R. Balzer, On the Inevitable intertwining of Specification and implementation, *Communication of the ACM*, Vol. 25 No. 7, July 1982, pp. 438-440. Also appears in *Software Specification techniques*, N. Gehani, A.D. McGettrick, Eds., 1986, pp. 41-45.

Deductive-Algorithmic Verification of Reactive Systems (Extended Abstract) *

Zohar Manna,
Nikolaj S. Bjørner, Anca Browne, Michael Colón, Bernd Finkbeiner,
Mark Pichora, Henny B. Sipma, and Tomás E. Uribe

Computer Science Department,
Stanford University
Stanford, CA. 94305-9045
manna@cs.stanford.edu

1 Introduction

Reactive systems have an ongoing interaction with their environment. Many systems can be seen as reactive systems, including computer hardware, concurrent programs, network protocols, and embedded systems. *Temporal logic* [Pnu77] is a convenient language for expressing properties of reactive systems. A *temporal verification methodology* provides procedures for proving that a given reactive system satisfies its temporal specification [MP95].

The two main approaches to the verification of temporal properties of reactive systems are deductive verification (*theorem-proving*) and algorithmic verification (*model checking*). In deductive verification, the validity of a temporal property over a given system is reduced to the general validity of first-order verification conditions. In algorithmic verification, a temporal property is established by an exhaustive search of the system's state space, looking for a counterexample computation.

Model checking procedures are usually automatic, while deductive verification often relies on user interaction to identify suitable lemmas and auxiliary assertions. However, model checking is usually applicable only to systems with a finite, fixed number of states, while the deductive approach can verify infinite-state systems and *parameterized* systems, where an unbounded number of similar components are composed.

*This research was supported in part by the National Science Foundation under grant CCR-95-27927, the Defense Advanced Research Projects Agency under NASA grant NAG2-892, ARO under grant DAAH04-95-1-0317, ARO under MURI grant DAAH04-96-1-0341, and by Army contract DABT63-96-C-0096 (DARPA).

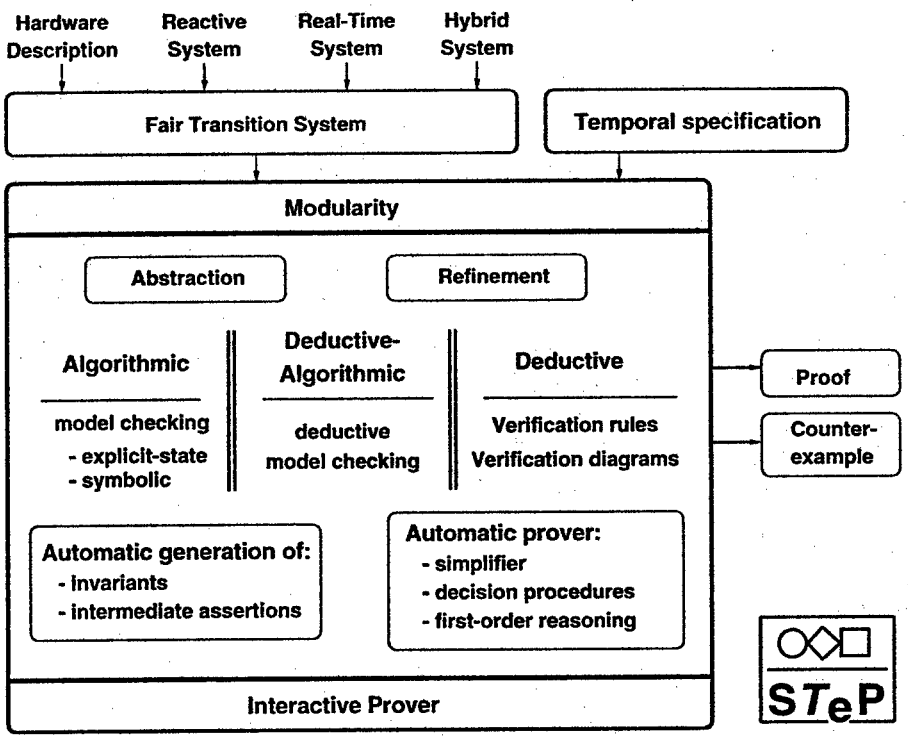


Figure 1: An outline of the STeP system

2 The STeP System

The Stanford Temporal Prover (STeP) supports the computer-aided formal verification of reactive, real time and hybrid systems based on their temporal specifications, expressed in linear-time temporal logic (LTL). STeP integrates model checking and deductive methods to allow the verification of a broad class of systems, including parameterized (N-component) circuit designs, parameterized (N-process) programs, and programs with infinite data domains.

Figure 1 presents an outline of the STeP system. The main inputs are a reactive system (which can be a hardware or software description, with real-time and hybrid components) and a property to be proven about the system, expressed as a temporal logic formula. Verification can be performed by model checking or deductive means, or a combination of the two.

The deductive methods of STeP verify temporal properties of systems by means of verification rules and verification diagrams. *Verification rules* are used to reduce temporal properties of systems to first-order verification conditions [MP95]. *Verification diagrams*

[MP94] provide a visual language for guiding, organizing, and displaying proofs, automatically generating the appropriate verification conditions as well.

On the algorithmic side, STeP features automatic explicit-state and BDD-based symbolic model checking for LTL. While the symbolic model checker is limited to finite-state systems, the explicit-state model checker can sometimes handle infinite-state ones, but is not guaranteed to terminate.

STeP also includes *deductive model checking* [SUM98], for the interactive model checking of infinite-state systems. Deductive model checking proceeds by transforming a diagram that abstracts the product of the system's state-space and the tableau (automaton) for the temporal property being verified.

STeP implements techniques for the *automatic generation of invariants* (and intermediate assertions) [BBM97]. STeP also provides an integrated suite of simplification and decision procedures for automatically checking the validity of a large class of first-order formulas (see Section 3.1).

Verification diagrams can be used to organize proofs that require user guidance. In all cases, the automatic prover is responsible for generating and proving the required verification conditions. An interactive Gentzen-style theorem prover is available to establish verification conditions that are not proved automatically. Tactics are available to automate parts of the high-level proof search by encoding long or repetitive sequences of proof commands.

Figure 2 describes the scope of STeP. Note that deductive methods allow the verification of real-time and hybrid systems whose discrete component is infinite-state (e.g. described by software, rather than a finite automaton). They are described by *clocked transition systems* (CTS) and *phase transition systems* (PTS), which generalize fair transition systems (see Section 3.2).

3 Recent Developments

3.1 Decision Procedures

The verification conditions generated in deductive verification refer to particular theories, reflecting the domain of computation of the system being verified. Rather than treat them as uninterpreted first-order formulas, *decision procedures* for the specific theories of interest can greatly increase the power, efficiency and ease of use of deductive verification systems.

STeP includes decision procedures for a number of theories common in formal verification: linear arithmetic, datatypes and finite domains occur in most systems to be verified; rationals and reals appear in the analysis of real-time and hybrid systems. As in most other verification tools with support for ground theory reasoning (e.g. [BDL96, ORR+96]), we have found congruence closure [NO80] to be an essential component of the decision procedures. Shostak's combination of decision procedures closure [Sho84, CLS96] improves the basic equality reasoning of congruence closure by efficiently integrating *solvable* theories such as inductive datatypes and linear arithmetic.

Decision procedures for bit-vectors [BP98] are particularly useful in the hardware domain. These are now part of STeP, together with a Verilog hardware description front-end.

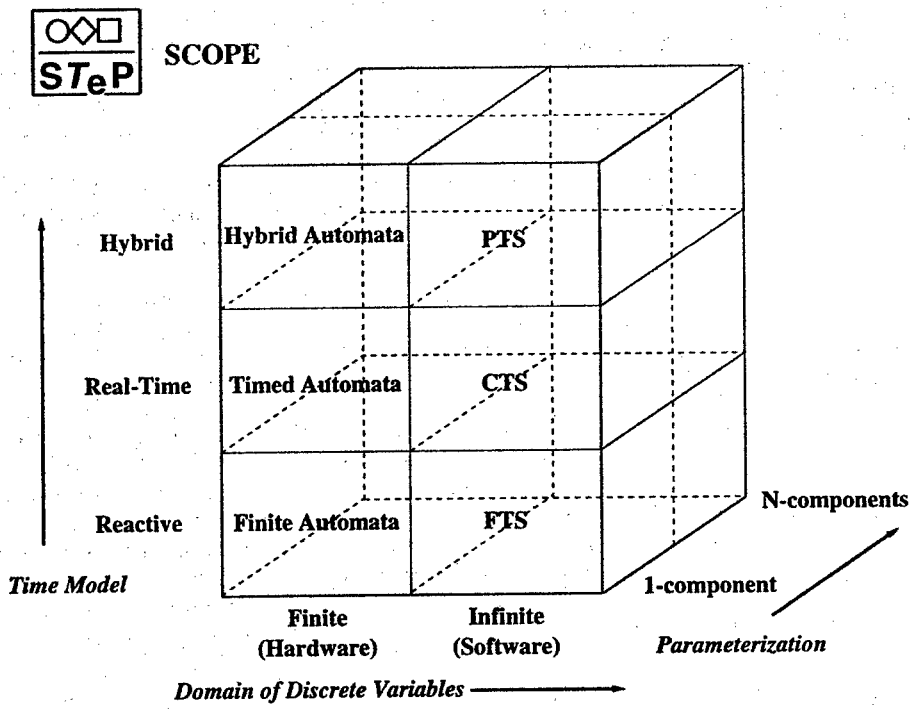


Figure 2: Scope of STeP

Quantifiers appear in the axiomatization of theories for which there is no specialized support. They are also present in verification conditions when verifying parameterized and (infinite-state) software, real-time and hybrid systems. Thus, the ground-level reasoning provided by most efficient decision procedures is no longer sufficient. To address this problem, STeP now includes methods for combining first-order reasoning and decision procedures [BSU97], in the form of a validity checker that performs partial quantifier instantiation based on rigid unification procedures.

3.2 Real-Time and Hybrid Systems

STeP supports the verification of safety properties of real-time and hybrid systems, based on the computational model of *clocked* and *phase transition systems* [MP96]. Systems described by timed transition systems, timed automata or hybrid automata can be readily translated into these formalisms.

In clocked transition systems, the untimed linear-time temporal logic is extended with a global clock measuring the overall progress of time. The transition system consists of

standard instantaneous transitions that can reset auxiliary clocks, and a transition that advances time, constrained by conditions on the global and auxiliary clocks. Phase transition systems contain other continuous variables, whose evolution is described by *activities*, usually in the form of differential equations. A *progress condition* limits the time that the system can stay in a particular discrete state.

The transition system model is retained by modeling the advance of time as a discrete transition parameterized by the duration of the time-step, and constrained by the progress condition. This representation allows STeP to reuse existing verification rules for untimed temporal logic. STeP has been applied to real-time benchmarks such as Fisher's mutual exclusion protocol and an N-process railroad crossing gate controller [HL94], as reported in [BMSU97]. Verification of hybrid systems with STeP is described in [MS98], including invariant generation methods and test cases.

3.3 Visual Verification

The interface for the latest version of STeP (2.0) is developed in Java, to allow the implementation of a wide class of visual verification formalisms. This includes deductive model checking [SUM98] and the generalized and hierarchical verification diagrams presented in [BMS95] and [BMS96]. These diagram-based verification formalisms share the following features [dAMSU97]:

- Diagrams are formal proof objects, which succinctly represent a number of verification conditions that replaces a set of textual verification rules.
- The verification conditions are *local* to the diagram; failed verification conditions point to missing edges or nodes, weak assertions, or possible bugs in the system. The necessary global properties of diagrams can be proved algorithmically.
- The construction of a diagram can be incremental, starting from a high-level outline and then filling in details as necessary. The diagrams for a given program can serve as documentation. They can also be re-used for similar proofs over refined or similar programs.

3.4 Modularity

STeP includes facilities for compositional specification and verification [FMS98]. Systems are described by a set of *modules*, which may be composed synchronously or asynchronously. Each module has an *interface* that determines the observability of module variables and transitions. Modular properties can be established by the same methods as global properties, accounting for environment transitions. Property inheritance then allows such properties to be used as lemmas in proofs over composite modules.

In [BMS96] we present *hierarchical verification diagrams*, which allow a proof by verification diagram to consist of multiple diagrams at different levels. In such diagrams, auxiliary temporal properties may be abstracted away and proved by lower-level diagrams.

To obtain STeP, send email to `step-request@cs.stanford.edu`. A new release, version 2.0, featuring most of the new interface and developments described above, will be made available by April 1998. A technical report describing the basic design of STeP is [MAB+94]. Recent test cases are reported in [BLM97] and [BMSU97], and a tutorial is presented in [BMSU98]. Information on the system can be also found on the web—see `http://theory.stanford.edu/~zm`.

References

[AH96] R. Alur and T.A. Henzinger, editors. *Proc. 8th Intl. Conference on Computer Aided Verification*, vol. 1102 of *LNCS*. Springer-Verlag, July 1996.

[BBM97] N.S. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, February 1997. Preliminary version appeared in 1st *Intl. Conf. on Principles and Practice of Constraint Programming*, vol. 976 of LNCS, pp. 589–623, Springer-Verlag, 1995.

[BDL96] C. Barrett, D.L. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *1st Intl. Conf. on Formal Methods in Computer-Aided Design*, vol. 1166 of *LNCS*, pages 187–201, November 1996.

[BLM97] N.S. Bjørner, U. Lerner, and Z. Manna. Deductive verification of parameterized fault-tolerant systems: A case study. In *Intl. Conf. on Temporal Logic*. Kluwer, 1997. To appear.

[BMS95] A. Browne, Z. Manna, and H.B. Sipma. Generalized temporal verification diagrams. In *15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, vol. 1026 of *LNCS*, pages 484–498. Springer-Verlag, 1995.

[BMS96] A. Browne, Z. Manna, and H.B. Sipma. Hierarchical verification using verification diagrams. In 2nd *Asian Computing Science Conf.*, vol. 1179 of *LNCS*, pages 276–286. Springer-Verlag, December 1996.

[BMSU97] N.S. Bjørner, Z. Manna, H.B. Sipma, and T.E. Uribe. Deductive verification of real-time systems using STeP. In *4th Intl. AMAST Workshop on Real-Time Systems*, vol. 1231 of *LNCS*, pages 22–43. Springer-Verlag, May 1997.

[BMSU98] N.S. Bjørner, Z. Manna, H.B. Sipma, and T.E. Uribe. *Verifying Temporal Properties of Reactive Systems: A STeP Tutorial.* Technical report, Computer Science Department, Stanford University, March 1998.

[BP98] N.S. Bjørner and M. Pichora. Deciding fixed and non-fixed size bit-vectors. In *Tools and Algorithms for the Construction of Systems (TACAS)*, LNCS. Springer-Verlag, 1998. To appear.

[BSU97] N.S. Bjørner, M.E. Stickel, and T.E. Uribe. A practical integration of first-order reasoning and decision procedures. In *Proc. of the 14th Intl. Conference on Automated Deduction*, vol. 1249 of *LNCS*, pages 101–115. Springer-Verlag, July 1997.

[CLS96] D. Cyrluk, P. Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In *Proc. of the* 13^{th} *Intl. Conference on Automated Deduction*, vol. 1104 of *LNCS*, pages 463–477. Springer-Verlag, 1996.

[dAMSU97] L. de Alfaro, Z. Manna, H.B. Sipma, and T.E. Uribe. Visual verification of reactive systems. In *Third Intl. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 97)*, vol. 1217 of *LNCS*, pages 334–350. Springer-Verlag, April 1997.

[FMS98] B. Finkbeiner, Z. Manna, and H.B. Sipma. Deductive verification of modular systems. In *International Symposium on Compositionality, COMPOS'97*, LNCS. Springer-Verlag, 1998. To appear.

[HL94] C. Heitmeyer and N. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *Proc. IEEE Real-Time Systems Symposium*, pages 120–131. IEEE Press, December 1994.

[MAB+94] Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E.S. Chang, M. Colón, L. de Alfaro, H. Devarajan, H.B. Sipma, and T.E. Uribe. STeP: The Stanford temporal prover. Technical Report STAN-CS-TR-94-1518, Computer Science Department, Stanford University, July 1994.

[MP94] Z. Manna and A. Pnueli. Temporal verification diagrams. In M. Hagiya and J.C. Mitchell, editors, *Proc. International Symposium on Theoretical Aspects of Computer Software*, vol. 789 of *LNCS*, pages 726–765. Springer-Verlag, 1994.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[MP96] Z. Manna and A. Pnueli. Clocked transition systems. Technical Report STAN-CS-TR-96-1566, Computer Science Department, Stanford University, April 1996.

[MS98] Z. Manna and H.B. Sipma. Deductive verification of hybrid systems using STeP. In T. Henzinger and S. Sastry, editors, *Hybrid Systems: Computation and Control*, LNCS. Springer-Verlag, 1998. To appear.

[NO80] G. Nelson and D.C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, April 1980.

[ORR+96] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Alur and Henzinger [AH96], pages 411–414.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Found. of Comp. Sci.*, pages 46–57. IEEE Computer Society Press, 1977.

[Sho84] R.E. Shostak. Deciding combinations of theories. *J. ACM*, 31(1):1–12, January 1984.

[SUM98] H.B. Sipma, T.E. Uribe, and Z. Manna. Deductive model checking. To appear in *Formal Methods in System Design*, 1998. Preliminary version appeared in *Proc.* 8^{th} *Intl. Conference on Computer Aided Verification*, vol. 1102 of *LNCS*, Springer-Verlag, pp. 208–219, 1996.

Reference Architectures and Conformance[†]

Sigurd Meldal

Computer Science Department
CalPoly

Computer Systems Laboratory
Stanford University

Email: smeldal@calpoly.edu

Abstract

We discuss the definition and modeling of reference architectures, and the notion of conformance. NSA's MISSI (Multilevel Information System Security Initiative) security reference architecture is used as an illustrative example.

We demonstrate that an ADL should have not only the capability to specify interfaces, connections and operational constraints, but also to specify how it is related (or conforms) with other architectures or to implementations.

A reference architecture such as MISSI is defined in Rapide [10] as a set of hierarchical interface connection architectures [9]. Each Rapide interface connection architecture serves as a reference architecture – an abstract architecture that allows a number of different implementations, but which enforces common structure and communication rules. The hierarchical reference architecture defines the MISSI policies at different levels – at the level of enclaves communicating through a network, at the level of each enclave being a local area network with firewalls and workstations and at the level of the individual workstations. The reference architecture identifies standard components, communication patterns and policies common to MISSI compliant networks of computer systems.

Key Words and Phrases: Software architectures, conformance, security, reference architecture, software engineering, specification, testing.

1. Introduction

Everybody knows what an *architecture* is – it is a set of *components* and *connections* between them. However, that is as far as agreement goes. What the proper methods of defining these entities are, what *conformance* means, what the distinctions are between an *architecture*, and *architecture style* and a *reference architecture*, these are issues that are unresolved (and presumably unresolvable, as they are questions closely related to world-views, methods and consequently often come down to pseudo-religious beliefs).

Architectures are used in different situations, and for distinct reasons. The most concrete use is in designing software systems, to make an initial sketch of it in terms of its module decomposition architecture in the top-down tradition of design, focusing on the high-level components and their means of interaction [24]. Architectures are also used to define *references* against which implementations can be checked for compliance. Such reference architectures define the functional components of the architecture and how the components may interact, but need not require that distinct components in the architecture necessarily be distinct also in the implementation. The use of reference architectures allows a separation of concerns in the system specification – distinct reference architectures address distinct aspects of the system (*e.g.*, there might be one reference architecture stating fault-tolerance requirements, another

[†] This project was funded by TRW under contract 23679HYL6M, DARPA under F30602–95–C–0277 (subcontract C-Q0097), and by NFR under contract 100426/410.

(such as the MISSI reference) stating security requirements, another (such as the ISO OSI reference stack) addressing communication protocols, etc.).

The presence of a component or connection between components in a reference architecture may signify different requirements, depending on which aspect of the system the reference addresses. *E.g.*, does the lack of a connection between two modules indicate a prohibition against their direct interaction (*i.e.*, is the interaction graph as given by the architecture supposed to be complete)? Does a connection between two components indicate that they *will* communicate (*i.e.* a connection represents not only a potential for interaction, it is also a requirement that such an interaction shall occur)? And in all cases, what is the concept of interaction anyway? Does an architecture imply what protocol an interaction shall adhere to? *E.g.* RPC vs. buffered pipes vs. passive, reactive systems vs. event broadcasting, etc.

In the end, what distinguishes one kind of architecture from another is the *conformance requirements* imposed by the architecture.

This article discusses how one can capture a security reference architecture in a manner amenable to analysis and automatic conformance checking. We shall start by pointing out in section 2 that the notion of *abstraction* changes when we move from *prescriptive* to *descriptive* specifications. This works well with the notion of conformance w.r.t. *multiple perspectives* (or *reference architectures)*, which we touch upon in section 3. Then, after giving a brief overview of the *Rapide* ADL in section 4, we present in section 6 the process of architecting using the *Rapide* ADL, giving examples from the MISSI reference architecture. In section 7 we go through all the top level requirements of the MISSI reference architecture one by one, showing how they are captured in the *Rapide* ADL. In section 8 we shall briefly look at how the reference architecture can be put to use for (semi-) automatic checking, visualization and analysis of implementation system conformance.

2. Abstract activity

Modern programming languages contain constructs for defining abstract *objects*. One of the consequences of "information hiding" is that an abstract object may accept many different implementations which are consistent with its abstract definition. Implementations may differ on the structures representing values, or the algorithms for the operations.

Similarly, the *activity* of a program, or system, may also be defined abstractly. On the one hand there is the operational abstraction embodied in the procedure and function concepts of most languages. There is also an abstraction mechanism inherent in the definition of *events* and *actors of interest* in a concurrent system. By identifying the classes of actors and activity we want to consider in describing the behavior of a system we establish a granularity of observation, ignoring details of implementation and the potentially composite nature of a single "event" or "actor."

In moving from procedural (imperative and state oriented) abstraction to behavioral (observational and event oriented) abstraction, a problem arises. A refinement of a procedural abstraction is accomplished by defining the abstraction in terms of other, more detailed procedural elements. This works well, since a procedural abstraction is *invoked* – low level activities are initiated by a higher level, the program invocation itself being the most abstract (Figure 1).

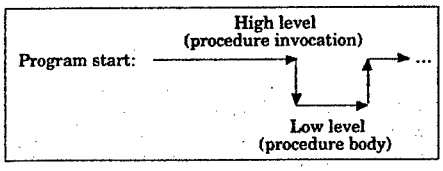


Figure 1:
Mapping from abstract to concrete activity

Furthermore, the identity of the invoker does not change as the program is refined, e.g. a high level procedure of a module and its invocation are retained in the final, fully detailed program.

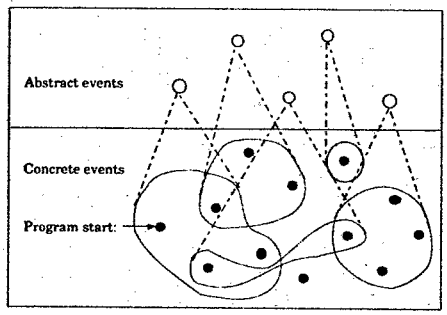


Figure 2:
Mapping from sets of concrete events to abstract events

The analogous observation does not necessarily hold true for an event based model. A process or event at some level of abstraction may not exist at a lower level. In specifying a lift system for instance, the concept of "lift" is natural, and a specification is readily given in terms of activities of such lifts. However, when implementing a lift system, there may be no distinct syntactical (or physical) entity corresponding to a particular lift of the abstract architecture (a lift being much more than simply the box itself – multiple lifts may share motors, sensor systems, etc.). The implementation of a lift may be in terms of motors, door sensors, arrays of buttons, etc., possibly shared among the abstract lifts. Events abstractly generated by a lift may be particular *patterns* of events at a level of increased detail. The abstract event of a lift moving from one floor to the next may correspond to the sequence of events "sense doors closed, signal controller, controller starts motor, sense reached next floor, signal controller, controller stops motor" in the implementation.

The result is a Copernican revolution: The causal relationship is one of concrete events and actors giving rise to more abstract ones (Figure 2).

3. Multiple Perspectives on a system

Consider a description of a hotel. In *describing* such an entity one might want to partition it in a number of different ways. One way could be according to domains – there is a domain of publicly accessible facilities, another of behind-the-scenes service facilities, etc. (Figure 3).

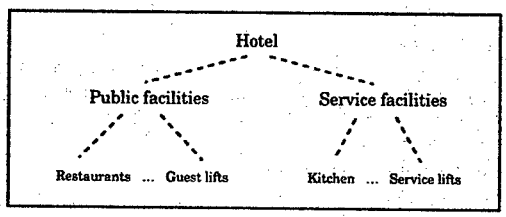


Figure 3: Conceptual decomposition

These domains may then be further subdivided, e.g. the public facilities one into restaurants, internal transportation (public lifts), etc., and the service facilities into kitchens, internal transportation (service lifts), etc., into the final, solid structure which is the implemented hotel.

In describing the functionality of the hotel, another decomposition may be more appropriate, for instance a partitioning of the hotel into domains of technical responsibility, e.g. electrical components, plumbing, etc. (Figure 4). This decomposition may also be refined through layers of less and less abstraction to the details of the finally implemented hotel.

Thinking procedurally, one has to choose one or the other of these views, the choice being determined primarily by expected ease of construction, i.e. by criteria not intrinsic to the system as perceived by the specifiers.

Having chosen a particular abstract description, it may function well as a component breakdown for construction purposes if *(1)* the specification language is *prescriptive* and *(2)* the flow of control in the system is initiated by imperatives at the highest level of abstraction.

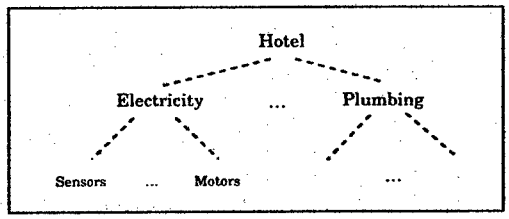


Figure 4: Functional decomposition

I.e. detailed activity occurs as a result of abstract activity being initiated, e.g. as in a procedure call (abstract) results in execution of the procedure body (detailed).

However, there is not always a single decomposition which is pre-eminent for a particular system. Furthermore, descriptive, event oriented concurrent languages appropriate for abstraction in concurrent systems do not satisfy the two criteria above, since an event at one abstraction level does not *cause* its component events at a more detailed level.

Moving from concrete architectures (such as the hotel above) to *software* architectures the discontinuity between architectures levels and between architectures and their implementation as a running system may become even more problematic, in terms of what conformance entails. Soni et al. [25] distinguishes between four architectural perspectives on a given system: The *conceptual* architecture, the *module interconnection* architecture, the *execution* architecture and the *code* architecture, and the transition from one to another may result in different identification of modules, connections etc., requiring a non-trivial definition of when (say) the module interconnection architecture conforms to a given conceptual architecture.

We seem to be caught on the horns of a dilemma. On the one hand, we need to be able to *describe* system behavior under different, often competing, perspectives. On the other hand, an architecture may also be used to *prescribe* behavior, indicating in some detail how the system shall generate the behavior the descriptions require.

4. An Architecture Definition Language

In reading an architecture description, the question of what the description actually *means* needs to be resolved unambiguously in the readers' and designers' mind in order to evaluate and then implement a given architecture. Without a clear understanding of the semantics of a notation (be it graphical – boxes and arrows – or textual) one cannot be sure that whatever is extracted from it (be it implementation strategies, modeling results, etc.) is implied by the description given, and understood by other readers of the architectural description.

An *interface connection architecture* [9] is defined by identifying

- *Components*: the primary elements of the architecture, and their means of interaction with other components. Components are considered black boxes constrained only by the definitions of their *interfaces.*
- *Connections*: the lines of interaction between components.
- *Conformance*: identifying minimum requirements of how an implementation may satisfy the architecture.

The *Rapide* model of architectures is *event based* – a basic notion being that architecture components are defined by the kinds of events they may generate or react to. An interface also identifies the semantics of a conforming component by giving event based *constraints,* specifying whether particular protocols are to be adhered to, identifying causal relationships between events, etc. Such constraints form the basis for analysis and testing tools, such as run-time checking for conformance violations [6, 17].

A successful ADL requires a high degree of flexibility in how an architecture can be refined. Naturally one wants to be able to refine interface definitions, making use of subtype substitutivity when extending an interface with new capabilities or by adding further constraints. In addition to this basic capability, an ADL should enable the definition of *hierar-*

chies of architectures, where one architecture can be interpreted quite flexibly as an implementation (or refinement) of another. The *Rapide map* construct gives the designer the tool to explicitly define how complex patterns of events in one architecture correspond to more abstract events of another, thereby enabling a powerful and checkable notion of *conformance*.

The literature presents a number of distinct ways of distinguishing *kinds* of architectures (*e.g.*, Soni et al. [25] makes a distinction between *object* and *function* decomposition architectures, among others, Shaw and Garlan [24] identifies *patterns* of object decomposition architectures). We prefer the notion that "an architecture description conveys a set of *views*, each of which depicts the system by describing domain concerns." [5] The distinction between different architectures descriptions then becomes one of a *difference of conformance requirements*. In moving from (say) a *module decomposition architecture* to an implementation, conformance would require disjoint sets of modules implementing distinct components of the architecture. In contrast, in checking whether a *reference architecture* is satisfied by a particular implementation one would make the weaker conformance requirement that there be a *mapping* of components and events at the implementation level to components and events of the reference architecture.

This perspective on what an architecture is allows a clean separation of concerns. One can specify multiple architectures for any given implementation, each focusing on a particular aspect of the system, each with an appropriate set of conformance requirements. For instance, when specifying a distributed object system it is reasonable to separate *security* concerns from *fault tolerance* concerns. Part of the security architecture for the system would state the conformance requirement that information should flow *only* along connections defined in the architecture; the architecture identifies the *maximal* connectivity of an information flow graph. In contrast, part of the fault tolerance architecture for the system would be to state the conformance requirement that information should be able to flow *independently* along all connections defined in the architecture, making no restrictions on the presence of extra connections; the architecture identifies the *minimal* connectivity of an information flow graph. In claiming that a particular implementation satisfies both perspectives the implementor would explicitly give the two maps, from the implementation to each of the reference architectures, showing the conformance argument.

The vocabulary of the *Rapide* ADL [10] incorporates and extends the basic vocabulary of interface connection architectures:

Events: Representing that something happened. What that something *is* may vary from architecture to architecture, and with varying degrees of abstraction.

Causality: In *Rapide* one can specify whether particular (patterns of) events should be independent or causally related. This allows a very precise description of information flow.

Patterns: Descriptions of how events may be related by causality, time or other relations. Patterns are described using an extension of regular expressions with placeholders to describe partial orders of events.

Constraints: Predicates, usually in the form of prescribed or proscribed patterns of behavior, indicating the intended functionality of a component.

Maps: Relating architectures to one another (and specifically, implementations to one or more architectures), indicating how conformance is obtained.

Rapide's object-oriented type- and module definition sublanguage provides features for code refinement and reuse (through inheritance and polymorphism) and specification refinement and reuse (through subtyping and polymorphism).

The semantic model of *Rapide* emphasizes causal and temporal relationships between events of a system, and thus provides the capability to be quite specific about how components of an architecture may (or may *not*) interact. Causal relations can often identify whether assumptions about the degrees of independence among an architecture's components are war-

ranted or not. *E.g.*, the focus on causal relationships allows the *Rapide* user to state in very general terms assumptions about the presence of covert channels, and to identify possible means of covert interaction in an architecture through the analysis of causal relationships displayed by test executions.

Furthermore, it allows tools to investigate the causal relations between events, distinguishing between temporal relationships that are causally significant and those that are not.

The *Rapide* pattern and constraint languages supports the definition of operational policies and specific protocols, which can take into account *causal-* as well as *time*-relationships between events.

The *Rapide* map construct supports explicit statements of conformance – the implementor of an architecture can state *exactly* how the implementation conforms: it defines which (sets of) components of the implementation play the role of particular components of the architecture, how patterns of events in the implementation correspond to more abstract events used in the architecture, etc. Since maps are given explicitly, they allow tools to check for conformance automatically, adding an extra degree of confidence that any conformance violations will be caught, offering a valuable supplement (or alternative) to formal reasoning.

The map construct is also a valuable tool whenever an architecture is given a *hierarchical structure*. *E.g.*, if one level of structure is defined in terms of federations of *enclaves* connected via *wide area networks*, and another level as network-connected *workstations*, *certificate servers*, etc., then maps are the means whereby the distinct levels can be related in the architecture definition. For instance, through the definition of appropriate maps the designer can identify how the set of networks, workstations and servers aggregate into enclaves and WANs.

5. Secure architectures

There are a number of perspectives one may apply when discussing the security aspects of a software architecture. In particular, in this document we shall address two aspects of the MISSI reference architecture:

Structures: That the secure architecture has a certain structure [24], requiring the existence of certain components (such as "certificate authorities," or "enclaves" [7]). The structures may be defined at different levels of abstraction, with different conformance requirements. We shall deal with

1. a *global* level, focusing on the main components and the overall constraints on their interaction. At this level general policies about information flow and the like may be stated, without regard to how these policy constraints are ensured by particular protocols, functional units, etc.
2. *a concept of operations* ("*conops*") level, focusing on the functional decomposition of the architecture, identifying the events of interest, the main functional components and their potential for interaction.
3. an *execution* level, describing the dynamic, modular decomposition structure of the system.

The architectures at each of these levels are related to one another and impose different conformance requirements on the implementation. Both the relationships and the conformance requirements must be defined.

Information flow integrity: That certain policies and procedures regarding the authorization and acceptability of information are adhered to as it is being generated and propagated. Such policies may be in terms of any of the three levels listed above and could also involve references to cryptographic and encoding requirements, as well.

6. The Architecting Process

The MISSI reference architecture is defined in a series of prose documents, some with first order predicate logic definitions of MISSI policies. In this exposition we shall stay with the overview document, given in full in [7]. The overview is an executive summary of the reference architecture, but contains enough detail to evaluate the utility of *Rapide* to specify the architecture.

We find the *process* of *constraints capture* in itself very useful. This process can be quite enlightening – interpreting the prose and giving it an unambiguous meaning often identifies potential contradictions or holes in the original definitions of the reference architecture. Even in the case where the final reference document is given in prose, we find that the exercise of formalizing the prose as it is being developed may help the development team, by enhancing their understanding of the interplay of their own statements.

Reference documents are also subject to mishaps, resulting from typographical mistakes through incomplete version-control to out-right conceptual misunderstandings. The sheer size of most such documents make them hard to check for consistency and correctness unless such checks are assisted by (semi-)automatic tools. Consequently, the presence of supporting tools should be almost mandatory in the definitions of standards. Tools require the existence of (parts of) the standard in a machine-manipulatable form, *i.e.*, in the form of a formalized set of definitions.

6.1 Prose and Constraints Capture

The process leading up to a formal capture of an architecture has three main steps: (1) identifying the components, (2) identifying how they are connected, and (3) identifying how the connections are used. The three steps are accompanied by a fourth, stating the conformance requirements, when relating the architecture to an implementation (or model, or a more detailed version). We'll go through the process of capturing the MISSI reference overview, giving examples of each of these steps.

Capturing the interface connection architectures defined in the MISSI specification, we first identify the *levels* of the reference architecture. In this article we shall deal with two levels, the *global* and the *concept of operations* levels (see section 5 above).

For each level we proceed to identify and define the *components* of the level by defining their interfaces (sections 6.2, 6.5.1), and then going on to define the *connections* among them (sections 6.3, 6.5.3) and how they are used (sections 6.4, 6.5.3)

As appropriate, we then go on to define how the components and activities of one level conform to those of another.

6.2 What are the components?

For each kind of component (such as an *enclave* at the global level) we define a *Rapide type*, whose interface is developed as the architecture is being refined. Part of this definition may identify how one type is a refinement or *subtype* of another [15]. Of course the interface definitions themselves rely on other types (such as *security classifications* and *security tokens*) already having been defined.

A very first approximation of an *enclave* type is given in Figure 5.

It identifies two key characteristics of an enclave:

1. The *provides* declaration of `s_class` makes it possible to refer to the security attributes (here exemplified by it having a security classification) of every enclave.
2. The *service* declaration of `wan_conn` states that every enclave interface contains a `Flow` entity which (as we shall see) defines the minimum communication capabilities of enclaves.

```
type Enclave is interface
provides
        s_class      : SecurityClassification;
service
        wan_conn     : Flow;
end Enclave;
```

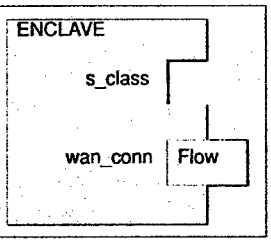


Figure 5: A definition of an enclave type

Architecture component interfaces can be highly structured. It may be helpful to think in terms of *plugs* and *sockets* [9]: a component's interface offers a set of distinguishable means of connecting it to its environment, similarly to what one expects in the hardware world. Such a means of connecting come in dual forms (as in *plugs* and *sockets* being duals in hardware), and may have further substructures (as in a single plug carrying pins/sockets for a number of wires).

It is natural to depict the Flow service type graphically (Figure 6), similarly to how we depict the Enclave interface definition in Figure 5. We can see that the wan_conn attribute has a structure; the declaration of its type, Flow, shows that wan_conn consists of two **action** declarations. An *out* action declaration indicates that the component may generate events which its environment may observe, an *in* action declaration indicates that the component may react to events generated by the environment. The wan_conn declaration is therefore in fact a bi-directional communication interface offering both a means of sending messages to the en-

```
type Flow is interface
action
   out  Release(data : Data; destination : Address);
   in   Accept (data : Data; destination : Address);
end Flow;

type DualFlow is interface
action
   in   Release(data : Data; destination : Address);
   out  Accept (data : Data; destination : Address);
end Flow;
```

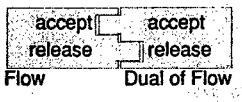


Figure 6: Plugs and sockets

vironment (intended to be a WAN) as well as of accepting such messages from the environment.

In *Rapide*, such structured communication interfaces are called *services*. The dual of the wan_conn service will be part of the interface of the wide area network component of the architecture, and is naturally depicted as the inverse of the Flow type (*i.e.*, it forms a plug to the Flows socket). Where the type Flow has an *out action* there will be a corresponding *in action* of the dual, and vice versa. One need not declare dual types explicitly, but can instead use the keyword dual. We have given the dual of Flow explicitly in Figure 6.

Though plausible as a first approximation in the global view of a distributed system, we may want to add some instrumentation points to the definition of an enclave. Consequently, in Figure 7 we create a subtype of the Enclave type. (Some of the other actions and functions will be used later. For each, the comment succeeding the declaration identifies where it is used.) We introduce a new out action called internal to be able to speak about things going on

```
type MISSI_Enclave is
   include Enclave;        - - Create a subtype of the
Enclave type
interface
provides
   function  release_reviewers        ()              - -
   (See rule 1(j, k))
            return set(release_reviewer_type);
action
   out internal             (a : Activity); - - (See Figure 10)
   out releasable           (d : Data);     - - (See Figure 17)
   out MISSI_releasable     (d : Data);     - - (See Figure 23)
   ...
end MISSI_Enclave;
```

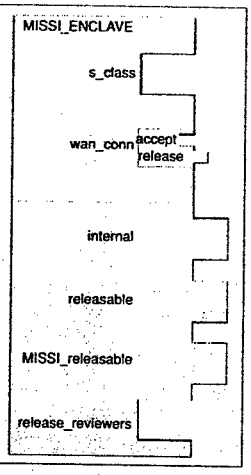


Figure 7: Extending the definition of an enclave

within the enclave (leaving the notion of "Activity" uninterpreted for now). As we shall see later, this turns out to allow an interesting architectural constraint about the existence of covert channels.

Having identified the *types* of components that make up the architecture, we define their number (if known), their structure (if any) and whether new components can be created while the system evolves, and whether existing components can terminate and remove themselves before the architecture terminates.

In the case of the MISSI reference architecture there is not much structure at the global level, and the architecture does not address the issue of dynamic component creation or removal. In its purest form, we may simply state that the components of the architecture are a *set* of enclaves, a single WAN (a simple routing model) and directory service agent and a set of unclassified (*i.e.*, non-DoD) sites, as in Figure 8.

This is deceptively simple, but then the architecture *is* rather simple, *at this level*. The complexity arises primarily at the lower level architecture, where we see a wide variety of architecture components and policies.

6.3 How are components connected? Adding structural constraints

Having identified the types and numbers of the components of the architecture, we proceed to define how they may interact. At this level of abstraction, the interaction is quite simple: The enclaves and sites are all connected to the WAN through their respective wan_conn services

```
architecture MISSI( ) is
   internet   : WAN;
   DNS        : DirectoryServiceAgent;
   enclaves   : set(MISSI_Enclave);
   sites      : set(Site);
   ...
end MISSI
```

Figure 8: The components of the MISSI reference architecture

```
connect
   for   e: Enclave in enclaves.enum() generate
      internet.socket to e.wan_conn;
end;
```

Figure 9: Connecting architecture components

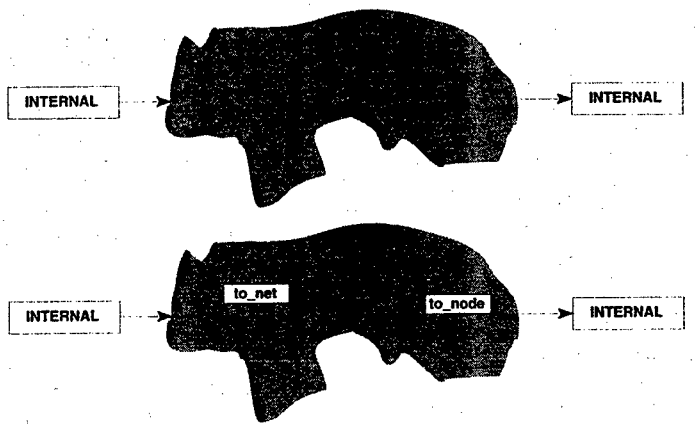
(Figure 9).

The role of the connection definitions are domain specific. In secure systems architectures, the interpretation of the set of connections would be that they identify *all* possible means of interaction among the architecture components. There is an implied frame axiom for the architecture specification that information shall flow only along those lines and in those forms explicitly defined by the connection definitions for the architecture (see Figure 10).

We notice that since all the enclaves are given a bi-directional connection to the internet, we have that the enclaves are all indirectly connected to each other. This is a common pattern – that components of an architecture communicate via intermediaries that allow for communication transformation, filtering, routing, etc. Such intermediaries are called *connectors*.

6.4 How are connections used? Adding operational constraints

After we have specified the structural properties of the global architecture, we go on to specify some *operational* requirements that implementations have to obey. Operational requirements define protocols and possibly other restrictions on the behavior of components of the




```
observe (?e1, ?e2: MISSI_Enclave)
   ?e1.internal → any()→ ?e2.internal where ?e1 ≠ ?e2
match
   ?e1.internal → any() → ?e1.wan_conn.to_net
      → any() →
   ?e2.wan_conn.to_node → ?e2.internal;
end;
```

Figure 10: A security constraint

architecture. Where a connection between two components indicates a *potential* for interaction, the operational specifications will indicate precisely under what circumstances such interaction actually can (or *must*) take place, as well as indicating when interaction shall *not* occur.

In the constraint sublanguage of the *Rapide* ADL one can specify simple protocols for interaction (such as handshaking, etc.), as well as more sophisticated requirements regarding information flow, causal relationships, etc. At the global level the most powerful security constraint would be that

No information should flow from one enclave to another without going through official network connections.

There are a number of different ways to make such a statement precise, and the *Rapide* formalization of the architecture specification allows us to clearly identify and thus discuss the alternatives. The strictest interpretation is probably that

There shall be no internal activity in two distinct enclaves such that they are causally related without intervening wan_conn *events.*

Stated in *Rapide* (see Figure 10), the semantics may be more immediately apparent: whenever we see a causal chain of events from an internal activity of one enclave to an internal activity of another enclave, then there must be two wan_conn events within that chain, one sending (from the originating enclave), and one receiving (at the other end). The variables *?e1*, *?e2* are free, indicating that the constraint holds for *all* enclaves.

This is a significantly stronger (and to-the-point) constraint than what we would obtain by stating the requirement in terms of *time*. If we interpreted "a → b" as "a happened before b in time" then the above constraint would be satisfied if two enclaves were (legitimately) interacting with high frequency while information were to flow covertly from the one to the other at a lower frequency. The fact that there would be legitimate wan_conn events interspersed between the sending and the receipt of covert information would legitimize the communication of the covert information. On the other hand, the interpretation of "→" as representing *causal* dependency correctly precludes such a scenario from being acceptable.

The *Rapide* pattern language has much in common with regular expressions extended with variables and the ability to evaluate Boolean expressions, and extended to deal with *partial orders* as well as the sequences of more traditional regular expressions. The key difference is that the *Rapide* pattern language encourages specifications of *causal dependency* relationships. The *Rapide* "a→b" relationship between two events requires that they occur in a particular order; a before b, and also that there be an established dependency between a and b, *e.g.* that a represents writing of data and b represents reading of that data, or a represents the sending of a message and b its receipt. For a full exposition of the *Rapide* pattern and constraint languages, see [11, 18, 19, 20]

6.5 Repeat as needed ... the *concept of operations* level

The next level of architecture is a *concept of operations* ("*conops*") architecture. The conops architecture specifies the structure of *enclaves*, and how the operations within an enclave are carried out by its various components (including human beings).

As with the global architecture, the definition of the conops architecture identifies (1) the components of an enclave, (2) their connections and (3) how these connections may (or may *not*) be used.

6.5.1 What are the components?

The components are such entities as *users* and *workstations*, *confidentiality* and *authentication servers* as well as other servers such as *firewalls*. We shall not enumerate all the component types of the conops architecture. However, the MISSI document [7] does give us an example

```
module certificate_authority
        (certificate_generator : Certificate_generator_type; ... )
    return Certificate_authority_type is
    ...
end;
```

Figure 11 An implicit architecture dependency

of a nontrivial decision we face when formalizing the definitions of the component types. It says:

2(a) "An authorized releaser for a particular enclave must be a MISSI certificate holder and reside within the enclave."

This paragraph introduces the component type "*authorized releaser,*" and can be interpreted in two different ways, depending on our interpretation of the word "must." If an authorized releaser *by definition* is a MISSI certificate holder, then one makes the type releaser a subtype of the type certificate_holder. A consequence of such a choice would be that one cannot entertain (or formally specify) situations where a releaser is *not* a certificate holder, just as one cannot entertain the notion that an even number not be an integer.

Another tack would be to identify the relationship between an enclave and its set of releasers, each of which is of the generic MISSI_user_type. In which case we are obliged to define a function from such user components to their set of certificates (in order to state that all releasers hold certificates) as well as a residency relation between enclaves and its residents (in order to state that the residency requirements should hold). Such functions and relations can be defined as being *part* of a component (*i.e.*, an attribute of it), or as a function or predicate external to the component. We chose the latter approach.

We are faced with a similar decision in paragraph 1(b):

1(b) "All legitimate MISSI users must have a valid certificate for some classification level they are cleared to read."

Is this a definition of what a "legitimate MISSI user" *is* (in which case we define the type legitimate_MISSI_user and add the requirement that the attribute certificate_set be non-empty)? Or is it a definition of when a MISSI-user is "*legitimate*" (in which case we define the type MISSI_user with the attribute legitimate, which is true if and only if the attribute certificate_set is non-empty)? We settled for the latter interpretation.

6.5.2 How are components connected?

At the enclave level we also see a number of requirements regarding access and connectivity, such as:

1(a) "Authorized certificate authorities (and no others) must be provided with access to certificate generation functions."

As with many of the MISSI requirements this one has both a prescriptive as well as a restrictive aspect: There shall be access for one class of components, and such access by any other component is prohibited. The former is reasonably interpreted as a structural requirement, the latter may either be structural (that there simply be no physical accessibility), or one of protocol (that there shall be no attempts at exercising the certificate generation functions without proper authorization.)

The prescriptive part of the requirement is easily modeled with in *Rapide* using interface type definitions (see Figure 12). The presence of a **requires** clause in the definition lists all the entities a Certificate_authority_type module expects to be able to use without further ado – it is up to the architecture implementation to supply it with a suitable module to satisfy this requirement. The **requires** section of a type specification indicates what the environment –

```
type Certificate_authority_type is
   include Authority
interface
provides
   function authorized () return Boolean;
requires
   certificate_generator : Certificate_generator_type;
   ...
end;
```

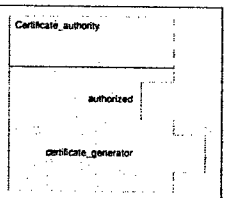


Figure 12: An enclave component identifying its requirements

the *architecture* – has to make available to objects of type Certificate_generator_type. This mechanism differs from the usual object-oriented approach of employing parameterization of the type or the object constructors of the type. If one were to employ the alternative of supplying the server references as parameters to object constructors as in Figure 11 then we would bury a key implicit element of the prose requirements; that the assignment of a server to a user is an *architectural* one, which may change over time as the system evolves and the user acquires or relinquishes certificates.

Rapide allows us to make the style distinction between parameterized *definitional dependencies* (which are identified by the parameter lists of type definitions), parameterized *implementation dependencies* (which are identified by the parameter lists of module constructors) and *(dynamic) architectural dependencies* (which are identified by **requires** sections in interface definitions).

The restrictive part of the requirement (*"...and no others..."*) can be addressed explicitly or implicitly. By using the frame axiom for security architecture conformance (*i.e.*, in the absence of any connections, no information flow shall take place) we can deduce this restriction from the absence of any explicit connections between modules that are *not* authorized certificate authorities and certificate generators. Such a *structure-oriented* representation of the requirement would be using conditional connections in the architecture itself to set up the connections for all the authorized certificate authorities (see Figure 13). Here the *architecture specification* makes clear that access to the new_token function will be given only to those

```
connect
   (?c : Certificate_authority_type)
      ?c.new_token where ?c.authorized
   to    certificate_generator.new_token;
```

Figure 13: A conditional connection

certificate_authority_type components that have the authorized attribute set to true.

However, a requirements document that relies on the *absence* of certain statements might be asking for too much of the reader.

If one instead wishes to make this requirement explicit in the formal version of the reference architecture then it is naturally rephrased as a *protocol requirement*; that all modules attempting to make use of the certificate generators are duly authorized. Since this is a usage restriction relevant to certificate generators, it is reasonable to locate it within the definition of the Certificate_generator interface (see Figure 14).

When it states *"Authorized certificate authorities (and no others)..."* the constraint interprets the *"(and no others)"* as meaning not only all non-authorized certificate authorities, but also all other entities of other categories. The mechanism is through observing *all* calls to the new_token function, and then requiring that all these calls be made by components of the

```
type Certificate is interface ... end;

type Certificate_generator_type is interface
     function new_token(...) return Certificate;
  ...
constraint
  observe (?p : root) new_token'call(performer is ?p)
  match (?c : Certificate_authority_type)
     new_token'call(performer is ?c) where ?c.authorized;
  end;
  ...
end certificate_generator_type;
```

Figure 14: A restrictive protocol definition

Certificate_authority_type, where that component also has the authorized attribute set to true.

A number of the requirements – 1(c,d,e) – as well as the later 1(e, g, h, i, k), are on the same form:

> *"All MISSI certificate holders must be provided with access to appropriate* <keyword> *functions for each classification level they are cleared to read."*

(Where the <*keyword*> identifies the distinct functions, such as *confidentiality*, *integrity*, and *certificate validation*.)

There are two elements to each of these requirements as well:

1. There is a reference to what a confidentiality (and similarly integrity-, certificate validation-, etc.) function *is*. That aspect deals with definitions of functions and abstract data types, and are best dealt with using an ADT- or object specification formalism. *Rapide* incorporates the data type specification capabilities of ANNA [8], but since the specification of datatypes impinges minimally on our discussion of architectures, we shall not pursue this aspect.
2. That for a particular functionality the actual function supplied may differ depending upon which access level is being exercised by the certificate holder. Consequently, access to server functions may change over time, as certificates are acquired or relinquished. Furthermore, there is no requirement that the appropriate function for a given access level be fixed for the duration of the system – consequently, the formalization should allow for a conforming system to supply different functions at different times for a given access level and user.

To state or allow for the latter is a challenge to ADLs and specification formalisms based on (first order) logics, which do not address the issue of *time*. In *Rapide* time is implicitly present throughout a specification, and can be made explicit as necessary through references to clocks or events.

We shall assume (see 1 above) that we can define precisely what is expected of a set of *confidentiality functions* (and similarly for the other functionalities).

Given such definitions of the server functions, we specify the access requirements explicitly (Figure 15). Each MISSI_user_type object will assume the (external) existence of a function returning a reference to a confidentiality server (assuming that the types Key_type, Wrap_info_type, and Wrapped_type are defined elsewhere), an integrity server and a validation server.

This requirement is formalized using the **requires** clause of *Rapide*. In so doing we signal that a MISSI_user_type object may call the function confidentiality_server with the expectation that the architecture (*i.e.*, the environment) will supply a binding for it. The archi-

```
type Confidentiality_ref is
        Confidentiality_server(Key_type, Wrap_info_type, Wrapped_type);
    - - and similarly for the other servers

type MISSI_user_type is interface
provides
    classification : set(Classification_type);
...
requires
    function confidentiality_server (c: Classification_type)
        return Confidentialy_ref
        constraint (classification.element(c));
    function integrity_server       (c: Classification_type)
        return Integrity_ref
        constraint (classification.element(c));
    function validation_server      (c: Classification_type)
        return Validation_ref
        constraint (classification.element(c));
    ...
end MISSI_user_type;
```

Figure 15: Capturing access requirements

tecture may change this binding during the execution of the system. By adding the "**constraint** (classification.element(c))" to the function declaration we identify that the function is only required and accessible for a particular classification level if the MISSI_user actually is cleared at that level.

The "*(and no others)*" part of requirements 1(j, k) are dynamic prohibitions and are formalized in the same way we made precise the similar injunction in 1(a), *i.e.*, as a check that whenever there is a call for a confidentiality_server it is from a component with the proper clearance.

6.5.3 How are connections used?

Finally, there are the policy requirements, stating preconditions for information flow within the enclave or from the enclave to the outside. An example is

2(c) *"All data transferred outside of a secret-high enclave and addressed to a MISSI certificate holder must be protected by a confidentiality service, a proof of origin non-repudiation service and a recipient authentication service."*

This can be modeled either as the data having certain properties (essentially having stamps of approval from the respective servers), or as a precondition on the *history* leading up to a release of data outside a secret-high enclave. We recommend the latter approach, in which case we make use of the *Rapide* pattern language to identify the protocol that defines a data release: it fits the pattern of Figure 16, *i.e.*, that for any piece of data, if it is released to the outside then that release has to be preceded by the three services checking it off.

```
pattern outside_release_ok(?d : data) is
        (conf_service(?d) ~ origin_service(?d) ~ recip_service(?d)) → data_release(?d)
end;
```

Figure 16: Abstracting patterns

```
map abstract_enclaves from e: enclave_architecture to MISSI_enclave is
rule
rule_1:
      (?e : event) ?e@
   ||> internal(?e);

rule_2:
      (?d : Data, ?a : Address) @firewall.wan_conn.to_net(?d,?a)
   ||> wan_conn.release(?d,?a);

rule_3:
      (  (?ws : COTSWorkstation; ?content : Data)
         ?ws.net_conn.to_node(Certificate_Validation, ?content)
      ~
         ?ws.net_conn.to_node(Integrity, ?content)
      ~
         ?ws.net_conn.to_node(Encryption,?content))
   ||>
      releasable(?content);;
   ...
end;
```

Figure 17: Three abstraction maps

6.6 Defining relationships between architectures

At this point in our process we have a definition of the global level of the reference architecture, whose principal components are MISSI_enclaves and WANs, and the conops level, whose principal components are workstations, firewalls, LANs, and servers.

Part of the definition of a reference architecture with multiple levels of abstraction identifies precisely how the levels are related. There are clear relationships between these two levels – *e.g.*, the enclave architectures of the lower level are modeling the MISSI_enclaves at the top level, the activities of the firewalls at one level represent release and accept events at the higher level, the simple wan_conn of the abstract enclave definition corresponds to the firewall_type objects of the conops architecture. But in the conops level definition there is no action "internal" which may play such a crucial role in the constraints of the global level architecture – the reference architecture must define what conops-level events correspond to the internal events of the global level.

It would not be a good idea to merge the definitions from the two levels into one unstructured definition of the notion of "enclave." Instead we use *Rapide maps* to relate components and activities of the conops architecture to their corresponding components and activities in the global architecture.

Figure 17 gives an example of such an abstraction map. It consists of three *rules*, each of which defines how occurrences of patterns of events at the conops level correspond to more abstract events at the global level.

The first rule indicates that *any* event in the conops enclave ("(?e : event) ?e@") will be mapped up to ("||>") the abstract internal event, indicating that something happened (but where we abstract away from the particulars of what happened). The second rule maps each transmission of data from the firewall to the WAN ("@firewall.wan_conn.to_net") to the abstract event release, representing the flow of information out of the enclave, abstracting away the particulars of how the information became public. The last rule is an example of how a more complex pattern of events may represent a single abstract event: Whenever a piece of information (represented by the placeholder ?content) has been approved by the validation, integrity, and encryption servers then the information becomes releasable, abstracting away from the actual protocol required for attaining this status.

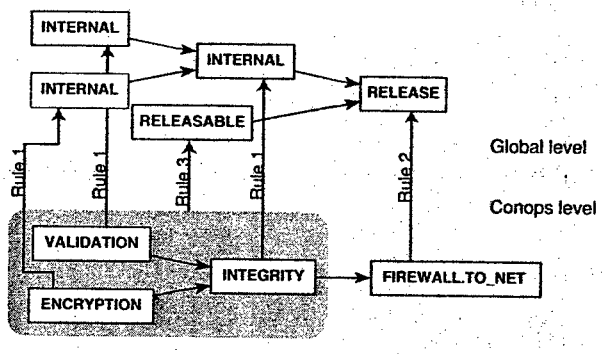


Figure 18: Events at two levels of architectural abstraction

Figure 18 shows an excerpt from a computation, indicating the two levels of abstraction and the relationship between a set of events at the lower level with a single abstract event at the higher.

As we see, there is no prohibition against a single concrete event participating in more than one abstract event (as each of the server events are both represented as abstract internal events as well as being part of the releasable event).

If one of the steps in the protocol is missing (for instance, if the Validation never took place), we would not get the required Releasable global event. The result would be as in Figure 19, and would result in a violation of a global level constraint.

7. Formalizing the MISSI requirements summary

In this section we go through all the requirements of the MISSI overview, showing how we would capture them in *Rapide*.

We have already dealt with the very first requirement (section 6.5.2):

I(a) "Authorized certificate authorities (and no others) must be provided with access to certificate generation functions."

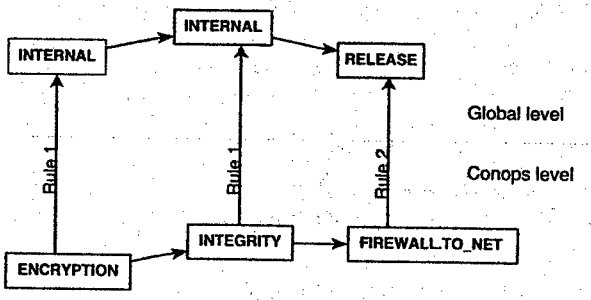


Figure 19: Missing conops event → missing global event

```
type MISSI_user_type is interface
provides
     function classification  () return set(Classification_type);
     function certificates () return set(Certificate_type);
     function legitimate      () return Boolean;
     ...
     function residency       () return Enclave;
     ...
constraint
     legitimate() = not certificates().empty;
     legitimate() implies
     not map(certificate_type,classification_type,certificates(),security_level).
                                                  intersect(classification()).empty;
end MISSI_user_type;
```

Figure 20 An invariant constraint

We have also touched upon the next requirement earlier (section 6.2):

1(b) "All legitimate MISSI users must have a valid certificate for some classification level they are cleared to read. Entities with valid certificates must be legitimate MISSI users."

If this is a definition of when a MISSI-user is "*legitimate*" we define the type MISSI_user with the attribute legitimate, which is true if and only if the attribute "certificate_set" is non-empty.[1]

The last constraint implies the first, of course, but in the interest of clarity of intention we state both explicitly, since redundancy adds rather than detracts from the confidence we have in the specification.

An alternative representation would define two types; MISSI_user_type and legit_MISSI_user_type <: MISSI_user_type. The latter would be constrained always to have in hand appropriate certificates, the former would allow its transformation into a legit_MISSI_user_type object after performing the appropriate checks.

The next three requirements – 1(c,d,e) – as well as the later 1(e, g, h, i, k), all contain a requirement on the same form:

"All MISSI certificate holders must be provided with access to appropriate <keyword> *functions for each classification level they are cleared to read."*

(Where the *<keyword>* identifies the distinct functions, such as *confidentiality, integrity,* and *certificate validation.*) They have been discussed extensively earlier, in section 6.5.3.

Requirements 1(j, k) strengthens the access requirements by adding that accessed functionality be

"...for the enclave in which they reside. (All <entities> *are MISSI certificate holders and reside in the enclaves in which they perform their task.)"*

---

[1] The polymorphic function map takes two types S and T (the source and target type), an object M of type set(S) and a function F with signature S→T, and returns an object of type set(T), each of whose elements is the result of applying F to some element of M. The function security_level is assumed to map certificates to security levels.

[2] Each shaded area represents a releasable event justifying the corresponding release event. There is an example of a single releasable justifying multiple releases, as well as a single release being justified by multiple releasable events.

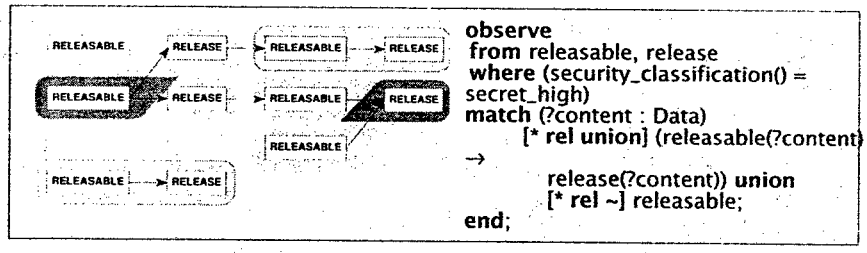


Figure 21: Satisfying the releasability requirement[2]

These are simply invariants over the relationships between components and enclaves, and could be stated in those terms, *e.g.*, in the subtype release_reviewer_type of the MISSI_user_type there is the invariant that:

```
...
not certificates().empty;
residency().release_reviewers().element(self);
...
```

Sections 2 and 3 of the requirement set identify the circumstances under which information may be released from or accepted into an enclave.

2(a) "An authorized releaser for a particular enclave must be a MISSI certificate holder and reside within the enclave."

2(a) is similar to the requirements of 1, and is dealt with in the same way.

2(b) "All data transferred outside of a secret-high enclave must have been sent by an authorized releaser in the originating enclave, must be protected by an integrity server, and must pass a releasability check in the originating enclave."

2(b) establishes protocol precursors for the event representing the release of data from an enclave. Assuming that data is being released by means of the firewall communicating to the network, the notion of data being *releasable* was captured earlier. Given that, 2(b) becomes a constraint of the abstract enclave definition. Observing release and releasability events (Figure 21), every communication to the net of a piece of data has to be preceded by a releasability event (but not the other way around – releasable data is not required to actually be released):

Note that there must be a *causal* chain from establishing releasability to the actual release.

The use of the **union** relation over the set of pairs of releasable and release events allows a single releasable event to justify multiple actual releases (as in Figure 21).

If the requirement specified that all releasable data actually be released then we would omit the second component of the union collecting all the dangling releasable events.

```
observe
    from MISSI_releasable, wan_conn.release
      where (security_classification() = secret_high)
    match (?content : Data)
            [* rel union] (MISSI_releasable(?content)
        →   wan_conn.release(?content))
      union [* rel ~] MISSI_releasable;
end;
```

Figure 22: MISSI releasability restriction

```
rule
   (?ws : COTSWorkstation; ?content : Data)
        (?ws.net_conn.to_node  (Confidentiality, ?content)
      ~
        ?ws.net_conn.to_node   (Non_repudiation, ?content)
      ~
        ?ws.net_conn.to_node   (Recipient_validation, ?content)
||>
      MISSI_releasable(?content);;
```

Figure 23: A variant on the releasability definition

```
observe
  (?content : Data; ?recipient : receipt_authentication_enclave; ?address : Address)
  wan_conn.release(?content, ?address)
          where (security_classification() = secret_high
             and ?recipient = ?address.enclave),
  →       ([* rel ~] receipt_acknowledge(?content.ack))
not match
  wan_conn.release;
end;
```

Figure 24: A negative form of constraint *2(d)*

2(c) "All data transferred outside of a secret-high enclave and addressed to a MISSI certificate holder must be protected by a confidentiality service, a proof of origin non-repudiation service, and a recipient authentication service."

2(c) is similarly structured to 2(b), the main difference being that we limit our interest to data addressed to MISSI certificate holders. By implication, this requires a global (specification) function mapping addresses to attributes of the addressee[3]. Figure 22 gives a variant on the 2(b) requirement. The global event MISSI_releasable is defined in Figure 23, and is similar to the definition of releasable (see Figure 17), as a mapping from a protocol pattern at the conops level to a single event at the global level. We assume that the function Recipient : Data→Root gives us the identity of the intended recipient of the data, and then use subtyping to limit the applicability of the mapping to those messages that have MISSI_users as recipients.

2(d) "If a recipient is capable of providing authentic receipts and the originator of the data requests a receipt, all data transferred outside of a secret-high enclave must be protected by a proof of receipt non-repudiation service."

This requirement mixes references to capabilities of enclaves (offering an authentication service) and events (the data being transferred with a return receipt request). To be "receipt confirmation capable" is modeled by adding a node Receipt_authentication_enclave to the type structure, introducing a subtype of the Enclave type. Stated in protocol terms, a receipt acknowledgment must be generated whenever data leaves a secret-high enclave addressed to a receipt confirmation capable component. There are a number of ways one can phrase this. As a negative, one can write that for each release event and all its (causally) subsequent acknowledgments for the receipt of the release, the set of acknowledgments cannot be empty (Figure 24).

[3] This mapping seems methodologically dubious, but it does not offer any problems for the transformation of the prose into precisely formalized requirements.

```
observe
    (?content : Data; ?recipient : Receipt_authentication_enclave; ?address :
Address)
    wan_conn.release(?content, ?address)
      where (security_classification() = secret_high
        and ?recipient = ?address.enclave),
      → ([* rel ~] receipt_acknowledge(?content.ack))
    match
      wan_conn.release → ([+ rel ~] receipt_acknowledge);
    end;
```

Figure 25: A positive form of constraint 2(d)

Or one can write it in positive terms – for each release event and all its (causally) subsequent acknowledgments for the receipt of the release, the set of acknowledgments has to contain at least one acknowledgment (Figure 25).

In both cases, the *Rapide* form is one of (1) filtering the set of events to extract those subsets (possibly overlapping) that are of interest (in this case to each single release and its (possibly empty) set of responding acknowledgments), and then (2) specifying the pattern these events have to comply with (in this case that the set of acknowledgments be non-empty).

3(a) "An authorized receiver for an enclave must be a MISSI certificate holders and reside within the enclave in question."

3(a) is similar to 2(a), and is dealt with in the same way.

3(b) "Any data admitted to a secret-high enclave from the outside must be protected by an integrity service, must pass an admissibility check for the enclave, and must have a designated recipient within the enclave who is authorized to receive external data."

3(b) is similar to 2(b), and is dealt with in the same way.

4(a) "All sensitive administrative data must be protected by an integrity service while in transit or in storage."

As with 2(b) and (c) there are two, quite distinct, perspective on this kind of constraint.

One can either view the requirements as related to *state*, *i.e.*, every piece of (administrative) data has some state attribute indicating whether it is in storage, in transit or in (possibly) other modes. In which case the natural mode of expression is one of first order logic (as in [7]), but at the cost of reduced checkability and increased complexity of expression – data and other basic types would acquire an ever-growing set of more or less obvious attributes, an attribute collection which may become intractable as the abstract notion of data becomes refined.

Or one can view it more dynamically, and focus on the *action* of storing or putting into transit a piece of data, in which case the assertion of being protected by an integrity service is tied to the transitional event itself. This is the path taken in the formalization of 2(b) and (c), and would be repeated for 4(a), here.

8. Putting a *Rapide* reference architecture to use

Given a *Rapide* formalization of the reference model we can put it to a number of different uses. The most obvious is as a precise definition of the model itself – being expressed in a formal language it allows us to draw unambiguous conclusions from the formalization based on testable arguments within a formal framework (in the case of *Rapide* constraints the framework is a simple one of sets and partial orders).

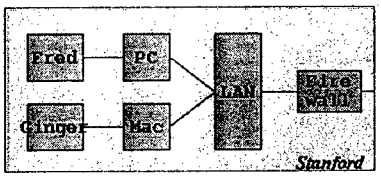


Figure 26: Some components of an intermediate level model architecture

Since *Rapide* is supported by a growing toolkit of visualization and testing modules [21, 22], the reference architecture can be the target for *conformance testing* by implementations purporting to satisfy the architecture's requirements. Such automatic conformance testing requires two things:

- An instrumentation of the implemented system which supplies the tools with the information required to compare the implementation to the reference architecture. Such an instrumentation can in many cases be automatically generated by a modified set of compilers,[4] generating the code necessary to create events and maintain the dependency graph.
- An abstraction *map* essentially defining how the patterns of events generated by the instrumentation correspond to the types of events and components referred to in the architecture.[5]

  Such a map makes the conformance argument precise, and adds documentation as to how the implementor thought her system relates to the reference architecture.

Given such instrumentation and the argument how conformance is obtained, the system conformance test becomes automatic, and can become a standard part of any regression test one might wish to subject the system to as its implementation evolves.

Furthermore, the instrumentation together with its conformance map can become an embedded, permanent part of the production system. The result is another layer of security checking, where the different perspective on the system offered by the conformance argument may detect architecture violations that might otherwise go unnoticed.

A variant of the conformance testing is the use of the tools for *scenario* testing and presentations. The *Rapide* toolkit has been applied to such diverse models as the SPARC V9 reference hardware architecture and a stock market model, as well as a simple scenario for security protocols based on elements of the MISSI reference architecture.

In the security model scenario we constructed a model vertically partitioned into three layers.

At the bottom layer we defined an executable conops model of *users*, *workstations*, *protocol servers*, *firewalls*, and *networks*.

The topology was one of a set of LANs, each with its workstations, firewalls and servers, and each workstation with its users. The LANs were connected by means of a WAN, through their respective firewall modules.

All the networks were broadcast networks.

This bottom layer corresponds to an actual system, a flat, relatively unorganized set of components communication hither and thither – possibly in conformance with the requirements of the reference architecture. Or possibly not – that is what the toolkit checks.

The second level is an intermediate one. Each architecture is an *enclave*, each of which is accompanied by a set of the enclave-related requirements (such as 2(b), about releasability). Each enclave in the intermediate architecture is the target for a *Rapide* map, which transforms patterns of conops model behaviors into activities defined for enclaves (*e.g.*, as in the definition of the releasability map, see Figure 17). Some components of an enclave is shown in

[4] Such an instrumented compiler-set exists for Java, Verilog and CORBA IDL besides for *Rapide* itself.

[5] We have already made use of such maps in defining how the abstract releasable event occurs as an abstraction from a pattern of lower-level events.

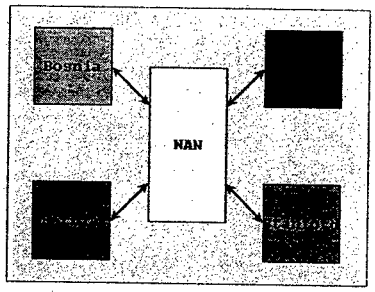


Figure 27: A global level architecture

Figure 26 (from an animation of the conformance check), an enclave with two users, two workstations, a LAN and a firewall (besides the local servers, not shown in this figure).

The third level is that of the *global* architecture, consisting of *enclaves, WANs,* etc. (Figure 27) gives an abbreviated view, from an animation of the reference architecture conformance test of a model with four enclaves.) At this level we check the constraints relating to multi-enclave concerns, such as the global requirement prohibiting covert channels. The architecture level can be obtained by maps directly from the conops model, or in two stages: by the maps from the conops model to the intermediate level, and then maps from the intermediate level on to the global level. Which of these one chooses is a question of whether the intermediate models contain all the information required for the global architecture model (*e.g.,* the notion of general internal activity) or not.

A model (or a system in testing or production) typically generates a large number of events. When investigating data for possible non-conformance it is critical that the number of data elements – events of possible interest – be reduced as early as possible. The *Rapide* toolkit offers two means to achieve this end. The first is the use of architecture maps in structuring the instrumentation. Each map construct results in the automatic construction of a *transformational filter* (or *sieve),* which passes on only those events that are considered significant in the abstraction, possibly transformed so as to aggregate event patterns into single events or simpler event patterns.

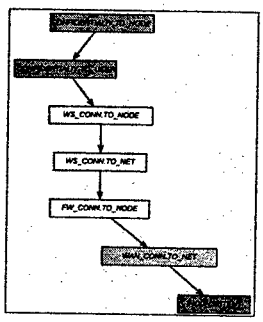


Figure 28: Detecting a protocol violation

The second is the visualization toolset of *Rapide.* This part of the toolset allows the user to apply various patterns of events to a given execution, displaying only those events fitting patterns of interest. Combined with the *Raptor* [22] animator this makes it possible to watch an animation of a running system at a chosen level of abstraction. Then, if interesting events (such as protocol violations) are detected, the user can move to the POV (*poset visualizer*) [21] and use it to investigate the causal patterns leading up to the events that piqued her interest. In particular, the POV allows the efficient removal of extraneous information, to ease the identification of interesting events among the clutter of all the events of the system.

As an example, consider the events of Figure 28. These were culled from the execution of a network model, after the occurrence of an *inconsistent* event was observed at the global level. (An *inconsistent* event signals the system's detection of a constraint violation, in this case the global releasability constraint of Figure 22). By moving from the global architecture to the conops architecture, using the POV, and then following the causal links past-wards from the *inconsistent* event, we identify its cause: the absence of the Integrity and Encryption steps of the protocol making a piece of information releasable. As the user only engaged the Confidentiality server, once the information was transmitted from the firewall to the WAN, she was in violation of the reference architecture constraints.

9. Conclusion

We have indicated how one may use the event based language of *Rapide* to capture elements of a reference architecture. Both the structural and the operational requirements of the archi-

tecture can be stated precisely in *Rapide*, and the resulting specification may become the basis for (1) analysis, (2) model checking, (3) implementation conformance testing and (4) production code conformance surveillance.

A key element in the successful application of an architecture description language to the design of reference or other software architectures is the degree to which it allows one to state *all* aspects of the architecture, and the flexibility of the *abstraction* mechanisms that may be applied when the conformance requirements are stated (as part of the architectural design). Distinct architectural perspectives require distinct abstraction mappings, and it is important that the designer be able to separate such perspectives from each other – giving separate reference architectures for each perspective, as appropriate.

Furthermore, an ADL is only as good as the tools that support it – in the absence of tool support, design capture and conformance reasoning easily devolves into vague hand-waving. The tool support should help automate conformance testing and other aspects of architecture design analysis, as well as allowing the designer to construct test scenarios and visualize the behavior of architecture conforming systems.

We have found that the *Rapide* ADL with its supporting toolset offers an interesting approach to the design of distributed architectures. In particular, the event orientation of the system, coupled with its sophisticated ability to identify causal chains and patterns of behaviors where causal relationships may play an integral role are quite enticing.

10. References

1. Allen, R., Garlan, D.: Formalizing architectural connection. In Proceedings of the Sixteenth International Conference on Software Engineering. IEEE Computer Society Press, May 1994.
2. Boehm, B. W.: Software Process Architectures. In *Proceedings of the First International Workshop on Architectures for Software Systems.* Seattle, WA, 1995. Published as CMU–CS–TR–95–151.
3. Garlan, D.: Research directions in software architectures. *ACM Computing Surveys,* 27(2): 257–261. 1995.
4. Garlan, D., Shaw, M.: *An Introduction to Software Architecture.* Volume I. World Scientific Publishing Company, 1993.
5. Ellis, W.J. et al.: Toward a Recommended Practice for Architectural Description. In *Proceedings 2nd IEEE International Conference on Engineering of Complex Computer Systems,* Montreal, Canada, 1996.
6. Gennart, B. A., Luckham, D. C.: Validating Discrete Event Simulations Using Pattern Mappings. In *Proceedings of the 29th Design Automation Conference (DAC)*, IEEE Computer Society Press, June 1992, pp. 414–419.
7. Johnson, D. R., Saydjari, F. F., Van Tassel, J. P.: MISSI security Policy: A Formal Approach. R2SPO Technical Report R2SPO–TR001–95, NSA/Central Security Service, July 1995.
8. Luckham, D. C.: *Programming with Specifications: An Introduction to ANNA, A Language for Specifying Ada Programs*, Springer-Verlag, Texts and Monographs in Computer Science, October, 1990.
9. Luckham, D. C., Vera, J., Meldal, S.: *Key Concepts in Architecture Definition Languages.* Submitted to the CACM. Also published as technical report CSL-TR-95-674, Stanford University, 1996.
10. Luckham, D.C., Vera, J.: An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(3):253–265, June 1993.

11. Luckham, D.C.: Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events, DIMACS Partial Order Methods Workshop IV, Princeton University, July 1996.
12. Meldal, S.: Supporting architecture mappings in concurrent systems design. In *Proceedings of the Australian Software Engineering Conference*. IREE Australia, May 1990.
13. Meszaros, G.: Software Architecture in BNR. In *Proceedings of the First International Workshop on Architectures for Software Systems*. Seattle, WA. 1995. Published as CMU–CS–TR–95–151.
14. Moriconi, M., Qian, X.: Correctness and composition of software architectures. In *Proceedings of ACM SIGSOFT'94: Symposium on Foundations of Software Engineering*. New Orleans, LA. December 1994.
15. Mitchell, J.C., Meldal, S., Madhav, N.: An Extension of Standard ML Modules with Subtyping and Inheritance. In *Proceedings of the 18th ACM Symp. on the Principles of Programming Languages,* ACM, ACM Press. 1991, pp. 270-278. Also published as Technical Report CSL-TR-91-472, Computer Systems Laboratory, Stanford University.
16. PAVG: The *Rapide* Architecture Description Language Reference Manual. <http://poset.stanford.edu/rapide/lrms/architectures.ps>
17. PAVG: *Rapide* toolset information. <http://poset.stanford.edu/rapide/tools.html>
18. PAVG: *Rapide* Examples. In preparation.
19. PAVG: The *Rapide* Pattern Language Reference Manual. <http://poset.stanford.edu/rapide/lrms/patterns.ps>
20. PAVG: The *Rapide* Constraint Language Reference Manual. In preparation.
21. PAVG: *POV–a partial order browser.* <http://poset.stanford.edu/rapide/tools-release.html>
22. PAVG: *Raptor–animating architecture models.* <http://poset.stanford.edu/rapide/tools-release.html>
23. Santoro, A., Park, W.: *SPARC-V9 architecture specification with Rapide.* Technical report CSL, Stanford University (to appear).
24. Shaw, M., Garlan, D.: *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.
25. Soni, D., Nord, R.L., Hofmeister, C.: Software Architecture in Industrial Applications. In *Proceedings of the 17th International Conference in Software Engineering.* ACM, April 1995.

Fine-grained and Structure-oriented Document Integration Tools are needed for Development Processes

*S. Gruner * / M. Nagl / A. Schürr*
Lehrstuhl für Informatik III, RWTH Aachen
Ahornstraße 55, (D)–52074 Aachen (Germany)
{stefan / nagl / andy}@i3.informatik.rwth–aachen.de

Abstract: Development processes of various engineering disciplines are usually rather complex. They consist of many interacting subprocesses, which are carried out by different developers. Each subprocess delivers its own documents, which are part of the overall result. All involved documents and their mutual, fine-grained dependencies are subject to permanent changes during the life-time of their development process. Keeping these documents in a consistent state is one of the most important prerequisites for the success of any engineering project. As completely automatic change control between documents is often impossible, interactive consistency monitoring and (re-)establishing tools are necessary, which we call integration tools. This paper reports about experiences in building integration tools for software engineering environments and about ongoing efforts to build similar integration tools for chemical process engineering. Furthermore, the paper presents an object-oriented and graph-grammar-based formal method for specifying integration tools and sketches how their implementations are derived from their high-level specifications.

Key words: development processes, product integration, tool specification, fine-grained interdocument relations, coupled graph grammars

1. Development Processes and their Results

Development Processes (DP) in areas such as software development, computer integrated manufacturing, or chemical process control usually involve different developers. Each developer produces a certain set of documents, which is part of the overall DP result. His documents have to be kept in a consistent state with documents produced by other developers. Between documents, directed and mutual consistency dependencies have to be taken into account. A software design specification, which depends on a requirement specification, is an example of a directed dependency. The different perspectives of a requirement specification —such as a data-oriented view and a function-oriented view— are an example for mutual document dependencies. Simultaneous engineering /BW 96/ aims at accelerating DPs by starting dependent subprocesses as early as possible with preliminary results (prereleases) of preceding subprocesses. Concurrent engineering /Re 93/, on the other hand, allows to develop different perspectives of the same product part in parallel.

1.1 Development Subprocesses and their Results

A key problem in the development of any engineering product is *change control*, especially in the case of simultaneous or concurrent engineering. Changes are carried out due to detected errors, due to changed design decisions but also, in the extreme case, due to changed requirements in an ongoing project. In the course of a usual development process many errors are made and the construction of required results is often not straightforward. Therefore, we can state that both development and maintenance of engineering products have to deal with permanent changes of intermediate or final DP results.

---

* *Stipendiat der Deutschen Forschungsgemeinschaft im Graduiertenkolleg für Informatik und Technik der Rheinisch-Westfälischen Technischen Hochschule Aachen*

In any application area mentioned above *complex document configurations* are built up and maintained. They do not only consist of the final configuration (e.g. the source code of a software system) but also of many further subconfigurations responsible for describing the requirements, the architectural plan, documenting the developed ideas and met decisions, assuring quality, or managing the whole development process. Such an overall configuration consists of many documents which, in turn, may have a complex inner structure.

Produced documents often have many fine-grained dependencies between their constituents. For the quality of the whole product and the efficiency of the total process these dependencies are of minor importance. A single developer is usually responsible for the internal consistency of a document. He should be able to keep all local consistency requirements in mind and, in many cases, he is supported by suitable document processing tools such as a syntax-directed diagram editor or a CASE analysis tool. The *fine-grained dependencies between documents* correspond to the interfaces between the work of different developers. As we shall see in section 2, no suitable support is available for keeping these interdocument dependencies in a consistent state. Therefore, we concentrate on this problem in the following. We use the relations between a requirement specification and the design of a software system as a running example.

For *coordinating a team of developers*, anagement information (administration configuration) about a project has to be built up and maintained. We distinguish between process, product (configuration and version), resources, and department or company information, and we regard their mutual relations in order to coordinate the labour of a team of developers. Management in this sense has to be supported by suitable tools. There, interesting problems arise as the administration configuration is changed in its structural form, when a development process is carried out /Kra 98/. However, in this paper we concentrate on the support of technical developers and the interfaces of their results.

Many DP documents, which are the results of technical development subprocesses, have a *semiformal contents*. Some documents are completely informal, as the nonfunctional requirements specification, where we find plain text possibly presented in a standardized (sub-)chapter form. Very few documents are formal, as e.g. a specification in a logic-based language such as Z. A standard case is that we find documents in diagrammatic, tabular, pictorial, or textual form which altogether possess an underlying structure and a more or less well-defined syntax. Examples of this kind are OO-analysis diagrams, module descriptions in a software design description language, and so on. So, "semiformal" either means that formalization of a certain DP result is not carried out completely or, even more, that the underlying document description language is only formalized to a certain extent.

Having most documents in a semiformal form, the *fine-grained relations* between documents *are semiformal, too*. Often, we can state that a certain increment (subpart) of one document may be related to an increment of another document if both increments are instances of corresponding types, have compatible properties, and appear in a certain context within their documents. A technical documentation may, for instance, contain a section for each module of a related software design document, and a section may contain a subsection for each exported resource of the related module.

1.2 Preserving Consistency of Dependent Documents

Language correspondences have to be elaborated, which define legal interdocument relations, before supporting tools can be built /Jan 92/. This is usually called *method integration* /Kro 93/. In some cases method integration rules require that instances of some type T_A in a document A are always related to instances of type some T_B in a document B, as it was the case with modules and sections in software design and technical documentation documents, respectively. This is

called a bijective (1:1) correspondence between increment types of documents. In many cases, we have (m:n) correspondences on the type level such that further information is needed in order to decide whether an increment of type T_A of document A may be related to an increment of type T_B in document B. Such a decision may depend on

- local properties of inspected increments,
- their contexts in the regarded pair of documents,
- and manual design decisions of an involved developer.

Usually, consistency establishing subprocesses of a development process *cannot be automated*. The development of some document B_i, which is the result of one subprocess, often depends on the result of another subprocess, some master document A, in a rather imprecisely defined way. Exceptions are generating an NC program from a CAD document, generating module frames from a software design document etc. In these cases, the contents of involved documents are closely related to each other and a complete formal definition of the corresponding interdocument dependencies is feasible.

The standard case is that *subprocesses are creative* in the sense that a developer is not able to come up with a precise and complete formal definition of a procedure (method, plan) how changes of a master document A have to be translated into corresponding changes of dependent documents B_i (cf. figure 1). As an example, regard the development of a coarse-grained software design for a given requirements specification. The design may be one of the "structured" world, an object-based or an object-oriented one /Nag 90/, and it is influenced by many factors such as the underlying middleware or the decision to (re-)use certain libraries or frameworks. As a consequence, there is no chance to automate the transition from requirements engineering to software design completely. However, the transition can be simplified by tools, which perform trivial subtasks on their own and keep track of once established relations between the requirements for and the design of a software system (see below).

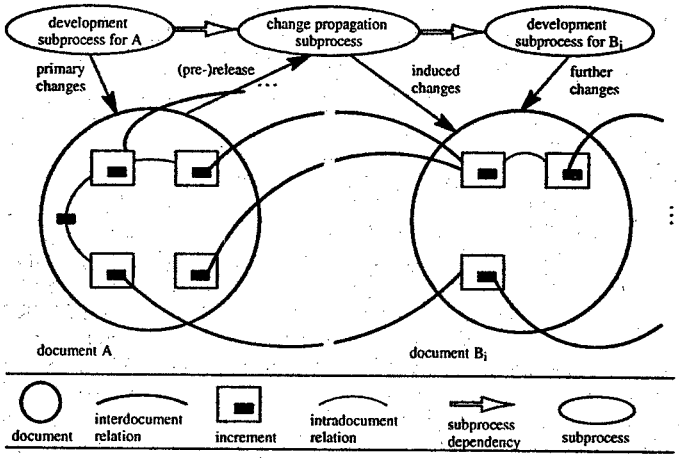


Fig. 1: *Dependent documents and their development subprocesses.*

Changes within a master document A require changes in a dependent document B_i, which then require further changes in dependent documents of B_i. There are many different possibilities

how to translate an update of a master document into updates of its dependent documents. Furthermore, rather different strategies may be used to propagate necessary changes along chains of dependent documents. One possibility is called *batch-oriented.* It reestablishes first complete consistency between a document A and all its directly dependent documents B_i w.r.t. a sequence of updates on A, before proceeding with the dependent documents of B_i. Another possibility is called *trace-oriented.* It propagates one performed update on a document A to all directly or indirectly affected dependent documents after the other. Both strategies have their specific advantages and disadvantages and should be supported by integration tools.

The following sections discuss the specification and realization of various types of *integration tools,* which are responsible for monitoring and (re-)establishing consistency of dependent documents. These tools have to regard the semiformal structure of corresponding documents. They have to give substantial support for fine-grained integration by regarding the current form of documents and offering different possibilities how to propagate changes, which are selected based on creative design decisions. Furthermore, they must not enforce certain orders of process steps as quite different consistency reestablishing strategies are possible. Finally, it should be possible to work with existing and a posteriori integrated tools, when manipulating the corresponding documents.

2. Available Support for Interdocument Consistency Control

The available support for monitoring and maintaining consistency of related documents on a fine-grained technical level is usually on a considerably lower level than the kind of tight integration sketched in the previous section. The standard procedure is that developers exchange documents in some low-level *standard format* (Postscript, SGML, HTML etc.). In any case, the developer of a dependent document has to find out which changes have taken place on a master document A and then to perform the right changes on the dependent documents B_i.

Another wide-spread approach, especially in the software engineering community /SB 93/, is to write *batch-oriented and automatically working converters* after the corresponding method integration has taken place. So, neither incremental changes of some master document A can be handled, nor do they regard that its dependent documents B_i are already elaborated to a certain state, nor can creative design decisions of developers be taken into account. Furthermore, such transformers are often hand-coded. As many documents in different languages as well as different method integration approaches exist, hand-coding of integration tools has to be replaced by generating them from high-level specifications.

A similar problem occurs with *document exchange standards* such as STEP with its data modeling language EXPRESS /ISO/ or CDIF /CDIF 94/. They define huge class diagrams (data models) for certain types of engineering documents, but disregard consistency relations between different types of documents to a great extent. The data modeling language EXPRESS allows, for instance, to define the data model of each type of documents as a separate module and to import the data model of one module into another one. Furthermore, EXPRESS offers rules for defining static integrity constraints across document boundaries. But it is very difficult to derive consistency establishing operations from these static integrity constraints. As a consequence, STEP/EXPRESS data models are not a suitable source of input for generating integration tools.

Also *hypertext systems,* as introduced in /Co 87/, do not offer an appropriate solution for preserving the consistency of a set of related documents. They have no knowledge about the semantics of links and just offer basic mechanisms to insert unspecific links and to browse along them. Consistency control is on a low level, namely detection of dangling links. Finally, all links

have to be established manually by the user. He has no profit if he is forced to insert them manually and then only gets warnings that a part of them are dangling.

More refined concepts can be found in (meta) software development environments, where documents are internally represented as *attributed syntax trees*. Support is given for propagating changed attribute values up and down the syntax tree. This allows to specify and generate analysis tools, which check interdocument consistency constraints /KKM 87/ if all regarded documents are modeled as subtrees of a common syntax tree. Other systems offer better support for the required nesting of documents. Gate nodes and door attributes of a distributed syntax tree model the transition from one document's language to another one /Bo 88/.

All syntax tree based systems mentioned above have problems with the specification of *active transformation tools* (in contrast to passive consistency checking tools). Attribute coupled grammars and variants thereof /GG 84, RT 88/, tree pattern matchers /AGT 89/, and context-sensitive tree transformation tools /CC 93/ are promising attempts to overcome these problems. They are useful for purposes like concrete syntax generation (unparsing) or compiler back-end generation. The still remaining problem with these approaches is that generated transformation tools are unidirectional, batch-oriented, and not interactive.

Federated database systems /SL 90/ represent another form of data integration. They offer a common global schema for different local database systems, which usually is used for retrieval of data but not for updates (update problem). Active database systems /DHW 95/ offer event-trigger mechanisms to keep databases consistent. Event-trigger mechanisms play about the same role for the data-oriented integration paradigm as messages do for the control-oriented integration paradigm of *broadcast message servers* /Rei 90, Fro 90, Fra 91/. Event-trigger mechanisms and broadcasted messages are still rather *low level means* to simplify the implementation of integration tools and to propagate updates between related documents or document processing tools.

For the sake of completeness we should mention the existence of tools, which coordinate development subprocesses and their results on a *coarse-grained level.*, i.e. without taking the internal structure of documents and the fine-grained interdocument relations between them into account. There, we find CAD frameworks /HN 90/ or EDM systems /McI 95/ for managing project databases. Workflow systems /GHS 95/ coordinate development subprocesses. Furthermore, we find configuration and version control tools on coarse- as well as fine-grained level. However, the latter are usually unspecific w.r.t. the structure of their documents /NW 98/.

3. Experiences in Building Integration Tools

Our experiences in building integration tools date back to 1988, when the implementation of our first integrated CASE tool prototype was finished /Lew 88/. Its most important integration tool has the task to keep a program's design in a consistent state with its technical documentation. This tool can be used in two different modes. Its *free format mode* allows to create arbitrary links between increments of the design document and sections or paragraphs of the accompanying technical documentation. In this case, the tool supports browsing along hyperlinks and issues warnings that once created hyperlinks are now dangling or that sources or targets of hypertext links are modified. The integration tool's more sophisticated *fixed format mode* enforces a structure of the technical documentation, which is closely related to the structure of its software design document. Any module of the design document corresponds to a section of the technical documentation, any exported module resource to a paragraph of the enclosing section. Section headlines are automatically derived from module names.

The (software design, technical documentation) integration tool realizes therefore a combination of a hyperlink *browser*, a consistency *checker*, and a consistency reestablishing *trans-*

formation tool (as any other integration tool presented here). Updates of the design document are immediately propagated as *change messages* to the dependent technical documentation. These messages are then asynchronously processed and—partly automatically, partly manually—translated into appropriate updates of the technical documentation. The integration tool was manually implemented without any kind of reuse of basic components and without a formal specification of its expected behavior. Needed hyperlinks were stored as pairs of unidirectional pointers in related documents.

Some years later an incrementally and automatically working (software design, Modula-2) integration tool was built using a more elaborate specification and implementation approach /Wes91/. First of all, the EBNF syntax definitions of both involved types of documents were related to each other and then manually translated into a *programmed graph rewriting specification* of the integration tool's functional behavior. Based on the graph rewriting specification, an integration tool was handcrafted, which keeps a software system's design and the corresponding configuration of Modula-2 implementation documents in a consistent state. The main progress of this integration tool—compared with the (software design, technical documentation) integrator—is that hyperlinks are now stored in a separate *integration document*. This allows the integration of already implemented types of documents more easily, simplifies multi-user access to related documents, and offers the appropriate database for storing information about an ongoing integration process.

The needs to build an integration tool, which keeps requirements engineering documents and software design documents in a consistent state, forced us to generalize the integration tool specification and implementation approach /Jan 92/. The most challenging feature of the (requirements engineering, software design) integration tool was the required interaction between the computation of applicable transformations rules and the manual selection of actually applied rules. This is due to the fact that consistency relations between Structured Analysis (SA) diagrams and Entity Relationship (ER) diagrams on one hand and design documents on the other hand are rather vague and context-dependent. An SA data flow diagram (DFD) may be translated into a module or a procedure of a design document depending on the number of applied occurrences of the DFD in its SA document, the way how related SA increments were already translated into software design increments, and creative design decisions of the involved integration tool user.

Figure 2 shows one example of an SA document and a graphical as well as a textual view of its related design document. The top left DFD SellProduct contains the three processes AcceptChoice, CheckPayment, and CalculatePrice as well as the two data stores CoinCharacteristics and PriceTable. The latter two provide needed input data for CheckPayment and CalculatePrice, respectively. The DFD in the left bottom corner of figure 2 represents the refinement of the process CheckPayment. All involved DFDs—except CheckPayment—are translated into so-called F(unction) modules, all data stores into so-called O(bject) modules, and the occurrence of a certain process or data store Y in a DFD X corresponds to an import between the related modules X and Y. The data flows between processes as well as the input and output ports of DFD SellProduct (the small rectangles labeled Input and Message on its left-hand side) are disregarded on this level of granularity. They may be useful later on for determining formal parameter lists of generated procedures.The missing module CheckPayment together with all its dependent components is just under construction. The black background color of DFD CheckPayment in the SA document informs the integration tool's user that this SA increment has not yet any counterpart in the corresponding program design. It is up to the user to decide whether it is worthwhile to build a separate

module for CheckPayment, too, or whether the functionality of CheckPayment may be implemented as a single local function of module SellProduct.

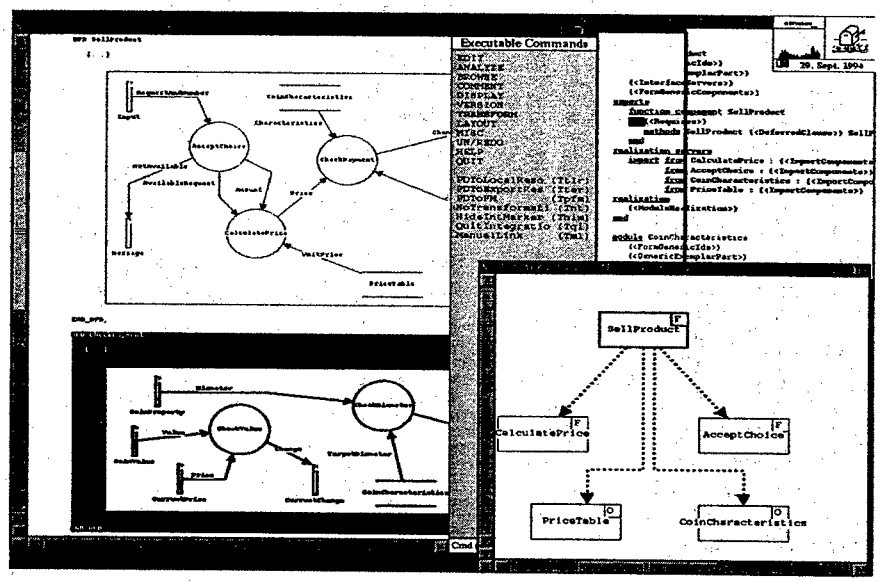


Fig. 2: *Requirements engineering and program design integration tool.*

The syntax of the regarded requirements engineering and software design documents with their diagrammatic notations was no longer defined in the form of EBNFs, but in the form of (extended) ER diagrams. As a consequence, a new *meta modeling approach* had to be invented to identify corresponding entity types of ER diagrams instead of corresponding nonterminal classes of previously used EBNFs. This approach to relate entity types of different (data base) schemas by deriving them from the same meta class has been adopted by the data base community for solving data base migration problems /JJ 95/.

Unfortunately, it is often not possible to derive consistency checking or document transformation code directly from constructed meta models. The problem is that entity type correspondences, such as DFD is either related to module or to procedure, are not precise enough to define the wanted behavior of integration tools. Therefore, we returned to the idea presented in /Wes91/ to describe the syntax of documents by means of grammars and to specify dependencies between documents by coupling these grammars. The main difference between the old proceeding in /Wes91/ and the new proceeding in /Lef 95/ is that the former one uses context-free string grammars (EBNFs) for this purpose, whereas the latter one is based on *context-sensitive graph grammars*, as already suggested in /Pra 71/.

Following this approach, the specification and realization of document integration tool proceeds now as follows:

1. The internal structures of dependent documents are modeled as *directed graphs*, which contain different classes (types) of attributed nodes and edges for different kinds of increments and intradocument relations.

2. UML[1] *class diagrams* are then constructed to define the relevant components of regarded document graphs from a static point of view.
3. Next, *correspondences* between UML class diagrams are established in accordance with the meta modeling approach of /Jan 92/. They identify possibly related increment classes (node types) of dependent documents.
4. Afterwards, *object diagrams* are used to define corresponding substructures (subgraphs) of related document graphs more precisely on the instance level.
5. These object diagrams are translated into *graph grammars* that generate those subsets of schema consistent document graphs, which consist of previously defined object diagrams only (thereby excluding intermediate inconsistent or incomplete document editing results as valid integration tool inputs).
6. Finally, the constructed *graph grammars are coupled* such that each production of a master document graph is related to a set of productions of a dependent document graph.

All needed kinds of integration tools may be derived from a single coupled graph grammar specification of the corresponding interdocument consistency relations. This includes a *forward transformation* tool, which propagates updates from a master document to the dependent document, a *reverse transformation* tool, which propagates updates from a dependent document back to its master document, or a pure *analysis* tool, which checks (traces) consistency of dependent documents without changing their contents.

Up to now, all integration tools are manually derived from a given coupled graph grammar specification, based on a *reusable framework* for the construction of integration tools /Nag 96/. The reusable framework offers, for instance, various forms of document traversing and comparing strategies as well as a standard implementation of integration documents. These integration documents, which were first introduced in /Wes 91/, are now used for storing all hyperlinks between two dependent documents together with all design decisions of users how to translate updates of master documents into updates of dependent documents.

It is the subject of ongoing research activities to translate coupled graph grammar specifications, which define interdocument consistency relations in a purely declarative manner, automatically into *programmed graph rewriting specifications*, which define the functional behavior of specific integration tools in the form of complex document graph transformations /JSZ 96/, /Gru97/. These graph rewriting specifications may be translated into equivalent C or Modula-2 code, using the compiler of the PROGRES graph grammar development environment /SWZ 95/.

The presented integration tool development process is explained in more detail in the following section 4. It was already successfully used for realizing another version of the (requirements engineering, software design) integration tool of figure 2 as well as for realizing new integration tools between software designs and Eiffel programs or between SA and ER diagrams /Lef 95/. Furthermore, related graph grammar based approaches were used for translating relational DBMS schemas into object-oriented DMBS schemas /JSZ 96/ and for integrating commercial SA editors with a research prototype for high-level timed Petri nets /BOP 97/. Last but not least a refined version of the presented integration approach is currently used for integrating chemical process engineering tools /NM 97/.

---

1. UML, the Unified Modeling Language, is the new standard notation for object–oriented analysis and design. It was developed by Rational Rose and is now an accepted standard of the Object Management Group OMG /FS 97/.

4. Specifying Interdocument Consistency Preserving Integration Tools

The previous section presented a number of graph grammar based integration tools and sketched their development. It is the purpose of this section to explain the graph grammar based development of integration tools in more detail, using the running example of the (requirements engineering, software design) integration tool. As already mentioned, the development of such a tool starts with modeling all involved types of documents as *directed graphs*. Different types of increments—such as DFD or Process—correspond to different types (classes) of nodes, different types of intradocument relations—such as DFD contains Process or DFD defines Process—are introduced as different types of directed edges (associations). Furthermore, node attributes are needed to represent local properties of increments, such as the Name of a DFD or a Process.

4.1 UML Class and Object Diagrams define Interdocument Relations

The needed components of document graphs and their relations are introduced as so-called *graph schemas*. It is a matter of taste whether an ER-diagram-like notation or an UML-class-diagram notation is used for this purpose. Within this paper, we prefer the upcoming standard OO notation UML, which allows to draw class diagrams as well object diagrams /FS 97/. Furthermore, UML offers the concept of packages, which allows to encapsulate document graph schemas and to distinguish between local and externally visible document graph components.

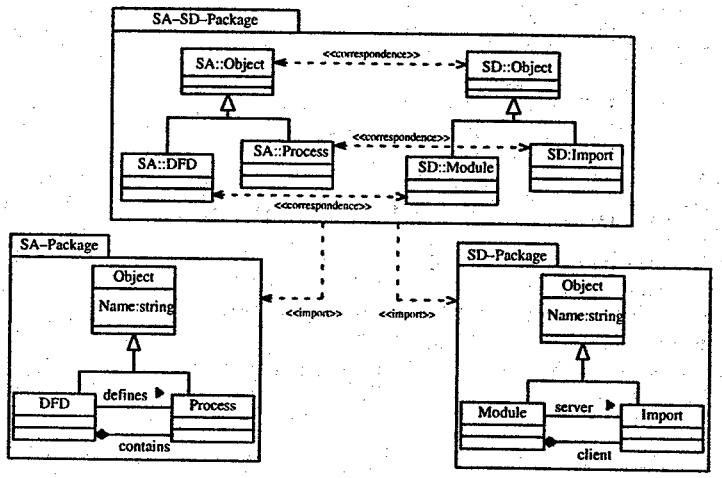


Fig. 3: *Corresponding SA and SD document graph schema definitions.*

Figure 3 shows a cut-out of the graph schema definitions for Structured Analysis (SA) and software design (SD) documents. Its packages SA and SD display only those definitions of classes and associations which are needed for the translation of a DFD and a Process into a Module and an Import, respectively. Both packages introduce a superclass Object to declare a Name attribute for DFD and Process on one side and for Module and Import on the other side. Please note that the Import relation is modeled as part of its client Module (source) and possesses the

Name of its server Module (target) as its own Name. This reflects the way how import relations (clauses) are defined in the textual software design document representation of figure 2.

The additional SA–SD package imports all externally visible classes of its dependent packages SA and SD and introduces the required *class correspondences* between the two (abstract) Object superclasses and between their subclasses DFD and Module as well as between their subclasses Process and Import. Many classes and class correspondences have been omitted in figure 4 due to lack of space, as e.g. the SD class Procedure and its correspondence to the SA class DFD.

Some *constraints* have been developed concerning legal and illegal combinations of generalization relationships with graph schema crossing correspondence relationships. These constraints, defined in /Gru 97/, prohibit e.g. the definition of a correspondence relationship between the classes SA::DFD and SD::Object in the presence of a correspondence relationship between the classes SA::Object and SD::Module. This is considered as a contradiction between the requirement that any SA::Object is mapped onto a SD::Module and the fact that a SA::DFD, a special kind of SA::Object, may be mapped onto any SD::Object.

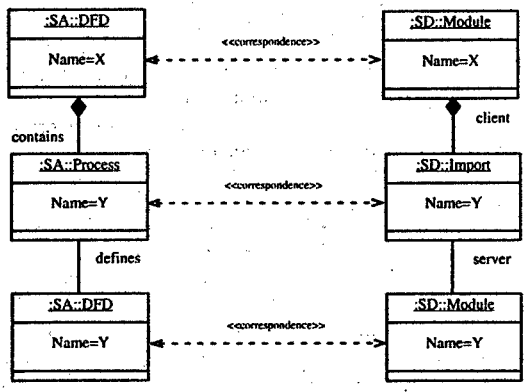


Fig. 4: *Object diagram definition of corresponding SA and SD subgraphs.*

Based on graph schema correspondences, which define a superset of all possible relations between SA increments and SD increments, it is now necessary to identify existing interdocument consistency relations more precisely. Experiences showed that *object diagrams* are the most appropriate notation for this purpose. They allow one to define pairs of subgraphs (subpatterns, substructures) on the instance level, which relate certain configurations of SA increments to corresponding configurations of SD increments and vice-versa. Figure 4 presents one example of this kind. It states that a DFD X, which contains a Process Y with its own DFD definition, may be related to a Module X, which contains an Import clause for Module Y. Furthermore, it requires that the DFD instances X and Y correspond to Module instances X and Y, and that the Process instance of Y in DFD X corresponds to the Import clause for Module Y in Module X.

Many object diagrams of this kind are necessary to define the set of all relevant interdocument relations between SA and SD document graphs. Unfortunately, it is not possible to establish useful *consistency or completeness criteria* for the resulting set of object diagrams in the general case. This is due to the fact that subgraphs of different object diagrams may overlap and that certain document subgraphs on one side may not have corresponding document subgraphs on the other side. Our running example requires e.g. the construction of another object diagram, which

relates the same SA subgraph as in figure 4 to a Module X, which contains a Procedure Y. As a consequence, it is not possible to interpret the constructed set of object diagrams as the consistent definition of a deterministic function, which translates SA documents into SD documents. The completeness criteria is violated by the fact that there may be some SA documents without corresponding SD documents. There is perhaps no rule how to translate a DFD X, which contains a Process X as a forbidden self-reference, into a corresponding SD document substructure.

4.2 From Class and Object Diagrams to Coupled Graph Grammars

The UML class diagrams and object diagrams of the previous subsection are the appropriate means to discuss the functionality of an integration tools with its future users. These users have the needed knowledge about the regarded application domain for building the appropriate set of class and object diagrams as well as for checking consistency and completeness of these diagrams. The following step of the integration tool building process is on a more technical level and may be performed without any assistance of application domain experts. It concerns the translation of a set of object diagrams into a *coupled graph grammar specification*. Each object diagram is translated into one or more coupled graph grammar productions. Each coupled graph grammar production is a pair of two regular graph grammar productions plus the definition of a correspondence relationship between the nodes on the left- and right-hand sides of the combined productions.

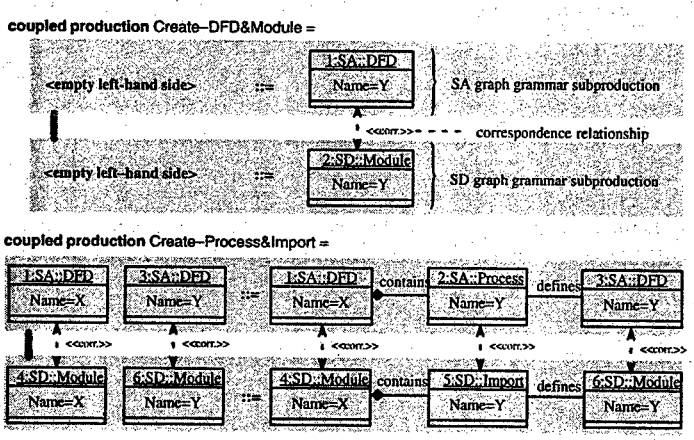


Fig. 5: *Two coupled graph grammar productions derived from figure 4.*

Figure 5 shows two examples of coupled graph grammar productions, which were produced by taking the object diagram of figure 4 as input. The main problem of the transition from object diagrams to coupled graph grammars is that we have to distinguish between *context nodes*, which are part of a production's left- and right-hand side, and *new nodes*, which are only part of a production's right-hand side. It is, for instance, not useful to define a graph grammar production, which creates a Process Y as part of a DFD X together with its DFD definition Y. As a consequence, we would not be able to deal with a DFD without any applied Process occurrences or with more than one occurrence. It is, therefore, better to translate the object diagram of figure 4 into two coupled graph grammar productions, as presented in figure 5. The first one, Create–DFD&Module, consists of two regular graph grammar subproductions, which have connected grey rectangles as background. Both subproductions have an empty left-hand side and a single

node on the right-hand side. The SA subproduction creates an isolated DFD node with Name = Y in the SA document graph, the SD subproduction a corresponding Module node with the same Name in the related SD document graph.

The following coupled graph grammar production Create–Process&Import of figure 5 matches two pairs of corresponding DFD and Module nodes with the left-hand sides of its two subproductions. These nodes together with their correspondence relationship are preserved by the given subproductions due to the fact that the defined right-hand sides contain their left-hand sides as subgraphs. Furthermore, the coupled subproductions create a new Process occurrence of DFD Y in DFD X as well as a corresponding Import clause with Module X as client and Module Y as server.

4.3 From Coupled Graph Grammars to Integration Tool Specifications

As already mentioned, one coupled graph grammar serves as the specification for a number of related but nevertheless quite differently behaving integration tools. The same coupled SA–SD graph grammar may for instance be used to develop an incrementally working forward engineering tool, which translates SA document updates into SD document updates, and a batch-oriented reverse engineering tool, which takes a complete SD document as input and produces a corresponding SA document as output. This a consequence of the fact that coupled graph grammar productions do not distinguish between master documents and dependent documents and that they do not prescribe how and when consistency between dependent documents is (re-)established.

Coupled graph grammars, which are constructed using a set of object diagrams as input, never contain node or edge deleting productions, i.e. the left-hand sides of their productions are always subgraphs of their right-hand sides. The restriction to non-deleting productions is not as severe as it seems to be at a first glance. This is due to the fact that the productions of a coupled graph grammar are *not the editing operations* for the involved types of documents. Document graph editing operations are defined without having certain interdocument consistency relations in mind. Furthermore, they do not only create, but modify and delete document components. Last but not least the editing operations for a certain type of document graphs may be used to manipulate subgraphs, which are irrelevant and therefore hidden for the regarded document integration task (cf. /Nag 96/ for further details concerning the definition of document views for integration tools).

The restriction to non-deleting productions is the necessary prerequisite for being able to derive *different kinds of (efficiently working) integration tools* from one coupled graph grammar specification. Otherwise, we would be forced to parse pairs of document graphs w.r.t. to unrestricted types of graph grammars in order to be able to check interdocument consistency relations (i.e. to solve the membership problem for type 0 grammars, which is undecidable in the general case). Having the restriction to non-deleting productions in mind, it is possible to translate coupled graph grammar productions into different sets of ordinary graph transformation rules. One set of transformation rules defines the functional behavior of a forward transformation integration tool, another one the behavior of the corresponding reverse transformation integration tool, a third one the behavior of a consistency checking tool, and so on.

Figure 6 shows one example of a derived SA–SD *forward transformation rule*. It translates a Process occurrence of DFD Y in DFD X of the SA document into an Import between the related Modules X and Y of the SD document. This rule was constructed as follows: its left-hand side is the combination of the right-hand side of the corresponding SA production with the left- and side of the corresponding SD production in figure 5, its right-hand side is the combination of the right-hand sides of the two coupled SA and SD productions.

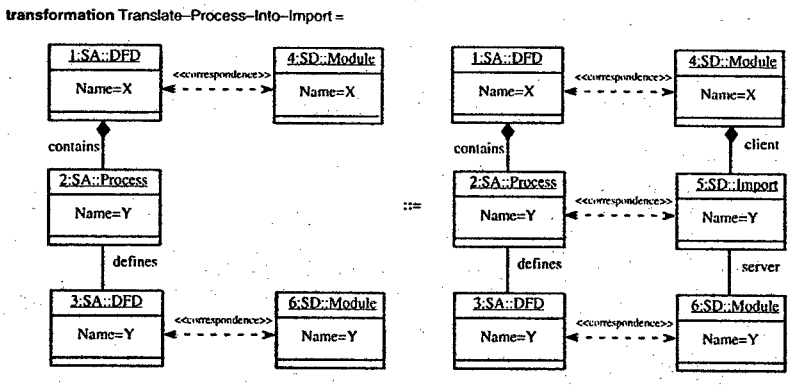


Fig. 6: *A forward graph transformation rule from SA to SD.*

An SD–SA *reverse transformation rule* may be constructed by simply exchanging the roles of SA and SD productions in the previous paragraph. A consistency checking and correspondence relationships establishing transformation rule may be built by merging the constructed forward and reverse transformation rules.

4.4 From Integration Tool Specifications to Implementations

All generated transformation rules have to be combined with a *reusable framework*, which determines the order of rule applications and which processes needed user interactions. Furthermore, the framework provides a certain bookkeeping strategy for not yet transformed document parts or already transformed but afterwards changed document parts. Please note that "real" graph transformation rules are more complex than the one presented in figure 6. They use a more sophisticated representation of correspondence relationships (as nodes and edges of separate integration document graphs) and manipulate therefore three related subgraphs of a hierarchical graph instead of one flat graph only /Nag 96/.

The finally needed translation from forward or backward graph transformation rules to *efficiently executable C or Modula-2 code* is supported by the PROGRES graph grammar environment /SWZ 95/. This environment is available as free software on the world-wide-web page

http://www–i3.informatik.rwth–aachen.de/research/progres/index.html

It represents an integrated set of tools for syntax-directed editing, analyzing, and executing single graph transformation rules or complex graph transformation programs. Two execution modes are supported: (1) direct interpretation of created specifications and (2) compilation into lower level programming languages such as C and Modula-2. Generated program fragments may be combined with a hand-crafted code, such as the integration tool framework mentioned above.

For further details concerning the construction of graph transformation rules from coupled graph grammars and the implementation of the needed framework the reader is referred to /Nag96/, /JSZ 96/.

5. Formal Background of Coupled Graph Grammars

The preceding sections introduced a graph grammar based method for the specification and implementation of document integration tools on a rather informal level. A complete formal defini-

tion of graph grammars, their underlying graph data models, and the definition of appropriate graph grammar coupling mechanisms is outside the scope of this paper. The interested reader is referred to /RS 96/ for a formal treatment of restricted types of *graph grammars* as a visual language syntax definition and parsing formalism and to the chapter 7 of the Handbook of Graph Grammars /Sch 97/ for the formal definition of a very general class of *programmed graph transformation systems*.

For further details concerning the usage of programmed graph transformation systems as a tool specification and implementation mechanism the reader is referred to /Nagl 96/. Further information concerning the design and implementation of the very high-level programming *language PROGRES* and its programming environment may be found in /SWZ 95/.

Last but not least the reader is referred to /Sch 94/ for a formal definition of *coupled graph grammars*, which is based on a very simply graph data model (without different types of nodes or edges and without attributes) and a simple form of graph grammar productions. A formal definition of coupled graph grammars for a more complex UML-compatible graph data model and more complex forms of productions is under development. Its first version, published in /Gru 97/, provides the formal background for the definition of correspondences between graph schemas (UML class diagrams) and the definition of coupled graph grammar productions, which respect the previously defined graph schema correspondences.

6. Summary and Future Work

Development processes of various engineering disciplines are usually rather complex. They consist of many interacting subprocesses, which are carried out by different developers. Various approaches are propagated nowadays how to support developers in executing their subprocesses and how to guarantee the overall consistency of their results, rather complex configurations of dependent technical documents. One may use the experience of developers

- by recording traces of successfully executed subprocesses and transforming them into repeatable process chunks /PDJ 94/,
- by offering means for direct multi-media communication, which are tightly integrated with technical document manipulating tools /Her 96/,
- and by realizing interactive integration tools, which help their users to monitor and (re-)establish interdocument consistency relations.

These three different approaches complement each other and may be combined with appropriate management tools to *support engineering processes* on a very high level /NW98/.

This paper had a main focus on *document integration tools*. It presented a graph grammar based method for deriving efficiently working integration tool implementations from very high-level interdocument consistency specifications. It is worth-while to notice that the presented method requires *not a complete formal specification* of the (semantics of the) considered types of documents and the involved modeling languages and methods. It is sufficient if all regarded documents have a well-defined internal structure (syntax definition) and if we have some knowledge about possibly corresponding patterns of increments in related documents.

The *functionality of presented integration tools* varies from a low-level hypertext editor, where all interdocument relations have to be created and maintained manually (if almost no knowledge about interdocument consistency relations is available) to an automatically working document transformation tool (if interdocument consistency relations may be defined as a total and deterministic function). In many cases the available knowledge about regarded document dependencies lies between these two extremes, such that the resulting integration tools are able to

perform trivial consistency (re-)establishing tasks on their own and to compute sets of possible consistency (re-)establishing actions in the general case.

It is the subject of ongoing research activities to generalize the presented coupled graph grammar formalism w.r.t. the form of permitted coupled subproductions. Furthermore, we are planning to realize a new generation of *graph grammar coupling tools*, based on the experiences reported in /JSZ 96/ with a rather ad hoc approach to use the PROGRES environment for this purpose. These tools will offer appropriate support for entering graph schema correspondences and coupled graph grammar productions, for checking the consistency between graph schema correspondences and coupled productions, and for translating coupled graph grammar specifications into ordinary PROGRES specifications of needed integration tools.

Finally, we are making our first experiences with changing our focus from tightly integrated software engineering environments to tightly *integrated chemical process engineering environments* /NM 97/. The new application domain forced us to complement coupled graph grammars with UML class and object diagrams as more appropriate means of communication between domain experts and future users of integration tools and their developers. But the main challenge of the new application domain is that we have to generalize the presented integration approach from the a-priori integration of self-developed (software engineering) tools to the *a-posteriori integration* of already existing chemical process engineering tools.

References

/AGT 89/ A. V. Aho / M. Ganapathi / S.W.K. Tijang: *Code Generation Using Tree Matching and Dynamic Programming*, TOPLAS 11, 4, 491–516 (1989).

/Bo 88/ P. Borras et al.: *Centaur: The System*, in P. Henderson, Proc. 3rd, ACM Softw. Eng. Symp. on Practical Software Development Environments, ACM Softw. Eng. Notes 13,5, 14–24 (1988).

/BOP 97/ L. Baresi / A. Orso / M. Pezzè: *Introducing Formal Specification Methods in Industrial Practice*, in: Proc. 19th In.t Conf. on Software Engineering (ICSE'19), IEEE Computer, 56–66 (1997).

/BW 96/ H.J. Bullinger / J. Warschatt (Eds.): *Concurrent Simultaneous Engineering Systems*, Berlin: Springer (1996).

/CC 93/ J.R. Cordy / J.H. Carmichael: *The TXL Programming Language Syntax and Informal Semantics Version 7*, Technical Report 93–355, Computing and Information Science at Queen's University, Kingston (1993).

/CDIF 94/ EIA / CDIF: *CDIF Family of Standards* (1994).

/Co 87/ J. Conklin: *Hypertext: An Introduction and Survey*, IEEE Computer, 17–41 (1987).

/DHW 95/ U. Dayal / E. Hanson / J. Widom: *Active Database Systems*, in Kim (Ed.): Modern Database Systems, 434–456, New York: ACM (1995).

/Fra 91/ B. Frankel: *The ToolTalk Service*, Sun Microsystems (1991).

/Fro 90/ D. Fromme: *HP Encapsulator: Bridging the Generation Gap*, HP–Journal, 59–68 (1990).

/FS 97/ M. Fowler / K. Scott: *UML Distilled*, New York: Addison Wesley (1997).

/GG 84/ H. Ganzinger / R. Giegerich: *Attribute Coupled Grammars*, in Proc. ACM Symp. on Compiler Construction, SIGPLAN Notices 17, 6, 172–184 (1984).

/GHS 95/ D. Georgakopoulos / M. Hornick / A. Sheth: *An Overview of Workflow Management: From Process Modelling to Workflow Automation Infrastructure*, Distributed and Parallel Databases 3, 119–153 (1995).

/Gru 97/ S. Gruner: *Schema Correspondences Support the Specification of Incremental Integration Tools based on pair graph grammars* (in German), TR AIB–97/5, RWTH Aachen (1997).

/Her 96/ O. Hermanns: *Multicast Communication in Cooperative Multimedia Systems* (in German), Diss. RWTH Aachen (1996).

/HN 90/ D. Harrison / A. Newton / R. Spickelmeir / T. Barnes: *Electronic CAD Frameworks*, Proc. IEEE 78–2, 393–419 (1990).

/ISO/ ISO CD 10303: *Product Data Representation and Exchange*, NIST Gaithersburg, USA.

/Jan 92/ Th. Janning: *Requirements Engineering and Programming in the Large* (in German), Diss. RWTH Aachen, Wiesbaden: Dt. Univ. Verlag (1992).

/JJ 95/ M. Jeusfeld / U. Johnen: *An Executable Metamodel For Re-Engineering of Database Schemas*, Int. Journal of Cooperative Information Systems, 4 (2/3), 237–258 (1995).

/JSZ 96/ J. Jahnke / W. Schäfer / A. Zündorf: *A Design Environment for Migrating Relational to Object-Oriented Data Base Systems*, in: Proc. Int. Conf. on Software Maintenance 1996, IEEE Computer Society Press, 163–170 (1996).

/KKM 87/ G. E. Kaiser / S. M. Kaplan / J. Micallef: *Multi-user, Distributed, Language- based Environment*, IEEE Software 4, 6, 58–67 (1987).

/Kra 98/ C. A. Krapp: *An Adaptable Environment for the Management of Development Processes*, Diss. RWTH Aachen, to appear.

/Kro 93/ K. Kronlöf: *Method Integration*, Chicester: Wiley (1993).

/Lef 95/ M. Lefering: *Integrating Tools in a Software Development Environment* (in German), Diss. RWTH Aachen, Aachen: Shaker (1995).

/Lew 88/ C. Lewerentz: *Interactive Design of Large Program Systems* (in German), Diss. RWTH Aachen, IFB 194, Berlin: Springer (1988).

/McI 95/ K.G. McIntosh: *Engineering Data Management – A Guide to Successful Implementation*, Maidenhead: McGraw-Hill (1995).

/Nag 90/ M. Nagl: *Software Engineering: Methodical Programming in the Large* (in German), Berlin: Springer (1990).

/Nag 96/ M. Nagl (Ed.): *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, LNCS 1170, Berlin: Springer (1996).

/NM 97/ M. Nag l /W. Marquardt: *SFB 476 IMPROVE: Support for Overlapping Developing Processes in Chemical Process Engineering* (in German), in Jarke/Pasedach/Pohl (Eds.): Informatik '97, Informatik aktuell, 143–154, Berlin: Springer (1997).

/NW 98/ M. Nagl / B. Westfechtel: *Integration of Development Systems in Engineering Applications* (in German), to appear, Berlin: Springer (1998).

/PDJ 94/ K. Pohl / R. Dömges / M. Jarke: *Decision-oriented Process Modelling*, Proc. 9th Intern. Software Process Workshop, ISPW 9, Airlie, Virginia (1994).

/Pra 71/ T.W. Pratt: *Pair Grammars, Graph Languages and String-to-Graph Translations*, Journal of Computer and System Sciences 5, 560–595 (1971).

/Re 93/ R. Reddy et al.: *Computer Support for Concurrent Engineering*, IEEE Computer 26–1, 12–16 (1993).

/Rei 90/ S. Reiss: *Interacting with the Field Environment*, Software–Practice and Experience 20–S1, 89–115, (1990).

/RS 96/ J. Rekers / A. Schürr: *Defining and Parsing Visual Languages with Layered Graph Grammars*, Journal of Visual Languages and Computing 7, 3 (1996).

/RT 88/ T. Reps / T. Teitelbaum: *The Synthesizer Generator Reference Manual*, New York: Springer (1988).

/SB 93/ D. Scheström/G. v. d. Broek (Eds.): *Tool Integration*, Chicester: Wiley (1993).

/Sch 94/ A. Schürr: *Specification of Graph Translators with Triple Graph Grammars*, in Tinhofer (Ed.): Proc. WG '94 Int. Workshop on Graph-Theoretic Concepts in Computer Science, LNCS 903, Berlin: Springer Verlag, 151–163 (1994).

/Sch 97/ A. Schürr: *Programmed Graph Replacement Systems*, in Rozenberg (Ed.): Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1, Singapore: World Scientific, 479–546 (1997).

/SL 90/ A. P. Sheth / J. A. Larson: *Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases*, Comp. Surveys 22–3, 183–236 (1990).

/SWZ 95/ A. Schürr / A.J. Winter / A. Zündorf: *Graph Grammar Engineering with PROGRES*, in Schäfer/Botella (Eds.): Proc. 5th ESEC, LNCS 989, 219–234 (1995).

/Wes 91/ B. Westfechtel: *Revision and Consistency Control in an Integrated Software Development Environment* (in German), Doct. Diss., RWTH Aachen, IFB 280 (1991).

Software and System Modeling Based on a Unified Formal Semantics*

M. Broy, F. Huber, B. Paech, B. Rumpe, K. Spies
Institut für Informatik, Technische Universität München
D-80333 München, Germany
email: {broy,huberf,paech,rumpe,spiesk}@informatik.tu-muenchen.de

Abstract

Modeling and documentation are two essential ingredients for the engineering discipline of software development. During the last twenty years a wide variety of description and modeling techniques as well as document formats has been proposed. However, often these are not integrated into a coherent methodology with well-defined dependencies between the models and documentations. This hampers focused software development, as well as the provision of powerful tool-support. In this paper we present the main issues and outline solutions in the direction of a unified, formal basis for software and system modeling.

1 Introduction

Computer technology for commercial applications has evolved rapidly from mainframes through personal computers to distributed systems. Software engineering has not been able to keep pace with the resulting demand on powerful application development methods. This is exemplified by an ever growing number of software projects running behind schedule, delivering faulty software, not meeting the users needs, or even failing completely. There are a number of reasons for that ranging from inadequate project management, over communication problems between domain experts and software developers to poorly documented and designed software. A recent inquiry on industrial software developers [DHP+98] has shown that despite the great variety of CASE-tools, development methods, and modeling techniques, software development still largely produces informal, incomplete and inconsistent requirements and design descriptions and poorly documented code. Modeling techniques are used selectively, but not integrated with each other and the coding. The great variety of proprietary modeling techniques and tools makes it difficult to choose an adequate selection for a project. As exemplified by the newly evolving standard

*The authors of this paper were funded by DFG-Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen", the project "SysLab" supported by DFG-Leibnitz and Siemens Nixdorf, and the Forschungsverbund ForSoft supported by the Bayerische Forschungsstiftung.

Unified Modeling Language [BRJ97], the techniques provide a rich collection of complex notations without the corresponding semantical foundation. Since only static models are linked to code, behavioural models can only serve as illustrations not worthwhile the big effort of building the model.

This situation will only change, if modeling techniques come with a set of development steps and tools for incremental model development, consistency checks, reasoning support and code generation. Mathematical description techniques like Z [Wor92] or LOTOS [Tur93] provide such development steps, but their uptake by industry is hampered by their heavy notation, lack of tools and lack of integration to established specification and assurance techniques [CGR93]. Recently, a number of approaches for the combination of mathematical and graphical modeling techniques have evolved (e.g. [Hußß97, BHH+97]) proving the viability of the integration of selected modeling techniques and formalisms. However, the integration of mathematical and graphical modeling techniques covering the whole process of system and software development is still an open problem.

The aim of this paper is to describe coherently the major issues in providing such an integrating basis. Experience on this subject has been gained mainly in the projects FOCUS [BDD+93], SYSLAB [BGH+97b] and AUTOFOCUS [HSS96]. The project FOCUS is devoted to developing a mathematical development method for distributed systems. SYSLAB concentrates on graphical description techniques, their formal semantics based on FOCUS and their methodical use, in particular for object-oriented systems. AUTOFOCUS is building a tool aimed at the development of distributed/embedded systems allowing the combined use of mathematical and graphical description techniques and providing powerful development steps based on the formal semantics. Its main application area are components of embedded systems. None of the projects covers the whole development process, but taken together they provide a clear picture of the road to follow.

The paper is structured as follows. In the first section we introduce FOCUS, the theory of stream processing functions, as the mathematical basis of our work. First, we present FOCUS independent of a particular application area. Then we show how to adapt it to object-oriented systems. FOCUS comes with a set of notations and a methodology for developing formal specifications which can only be touched on in this paper. Refinement and compositionality provide the foundation for the formal development steps. We close this section with a discussion on the enhancement of formal notations to be useful for practitioners.

We then go on to describe the *indirect* use of FOCUS as the integrating formal semantics for graphical modeling techniques used in software development. We describe a bunch of graphical description techniques covering the main system aspects. These modeling techniques are similar to the ones used by structured or object-oriented methods. However, they differ in detail, because they have been developed with a particular focus on an integrating formal semantics. The aim of that section is to make explicit the most important issues in providing such an integrating formal semantics.

The indirect use of formal methods is very valuable to the method developer. It is only useful to the system developer, if the modeling techniques are accompanied by powerful development steps which allow them to check and enforce the formal dependencies between the models. In the third section we discuss consistency checking, model validation and transformation as the most important development steps, together with possible tool support.

The modeling techniques and development steps have to be integrated into a process of system development, covering requirements definition, analysis, design and implementation. In the fourth section we present a framework making explicit the different modeling areas to be covered, namely the application domain, the system usage and the software system, as well as the interplay between different system views and their corresponding modeling techniques.
We close with an outlook on future work. Related work is dicussed along the way.

2 Semantic Framework

In this section we describe the formal semantical basis. First, we sketch the mathematics of system description, treating object-oriented systems as a special case. Then we present refinement as major constitutent of formal system development. We describe the process of formal system development and close with an evaluation of this direct use of FOCUS, a general framework for formal development of distributed reactive systems.

2.1 Mathematical Basics

FOCUS incorporates a variety of techniques, specification formalisms and semantic choices. We only give a short and informal description of the main concepts and some simple formulas. The interested reader is referred to details on [BS97, BDD+93] for an introduction and more formalization, and [BBSS97] for an overview of case studies. There are many other formal development methods and description techniques, see e.g. TLA, UNITY or PROCOS. For further reading and a comparison between these and many other formal methods like algebraic or temporal logic approaches based on an uniform example we refer to [BMS96].
According to the concept of FOCUS, a distributed system consists of a number of components that are partially connected with each other or with the environment via asynchronous one-way communication channels, comparable with unbounded FIFO-buffers. With the behaviours of components and the topology of the network – the connection of components via the communication channels – the system is completely described. The behaviour of a system can be deduced from the behaviour of its constituents. This is possible because the formal basis of FOCUS allows *modular* systems specification with *compositional* semantics.

Timed Streams

The basic data structure needed for the definition of component behaviour are *timed streams*. Assuming a global and discrete time we model the time flow by a special time signal called *time tick*. Denoted by $\surd$, a tick indicates the end of a time interval. A timed stream is a sequence of messages and $\surd$ that contain an infinite number of time ticks. Apart from the time ticks the stream may contain a finite or infinite number of messages. Let M be a set of messages that does not contain the time signal $\surd$. By M^ω we denote streams of messages and by $M^{\bar{\omega}}$ we denote the set of infinite, timed streams which contain an infinite number of ticks. To illustrate the concept of a timed stream we show a simple example. The timed stream

$$a \surd ab \surd \surd bca \surd b \surd \ldots$$

contains the stream of small letters *aabbcab*. In the first time intervall a is communicated, in the third intervall there is no communication and in the fourth intervall first b then c and last a is communicated.
The special time signal $\surd$ should not be understood as a message that is transmitted, but as a semantic concept to represent discrete global time. With timed streams complete communication histories are modelled: a specific stream that is associated with a channel between two components contains all information about *what* message is sent *when* between these components. Semantic variants of FOCUS abstract from time in the *untimed model*. In the *synchronous model* in every time intervall at most one message can be transported between two components.

Component Definition

A (system) component is an active information processing unit that communicates with its environment through a set of input and output channels. To define a component, first the *interface* must be declared. This contains a description of its input and output channels as well as the type of messages that can be received or sent via these channels. The *behaviour* of a component is described by a relation between its input streams and its output streams, containing the set of communication histories that are valid for this component. One way to describe this relation is to define a *stream-processing function* that maps input streams to sets of output streams. This function reads an input stream message by message, and writes - as reaction - some output messages onto the output channels. Stream-processing functions have to fulfill further semantic properties as continuity, realizability, time-guardedness and more, as explained in literature. It is possible to use state parameters to store control states or additional data that can be helpful for easier modelling.
Let I be the set of input channels and O the be the set of output channels. Then by (I, O) the *syntactic interface* of a component is given. With every channel in $I \cup O$ we associate a data type indicating the type of messages sent on that channel.
To describe and to design the topology and the behaviour of a distributed system and its components, FOCUS offers different graphical and diagrammatical notations, see section 3. All these description formalisms are well founded in the mathematical framework described in this section. A graphical representation of a component with its syntactic interface $I = \{i_1, \ldots, i_n\}$ resp. $O = \{o_1, \ldots, o_m\}$ and the individual channel types $S_1, \ldots, S_n$ resp. $R_1, \ldots, R_m$ is shown in Figure 1.

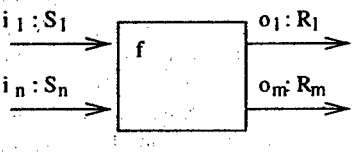


Figure 1: Graphical Representation of a Component as Dataflow Node

Given a set of channels C we denote by $\vec{C}$ the set of all channel valuations. It is defined by:

$$\vec{C} = (C \to M^{\overline{\omega}})$$

Channel valuations are the assignments of timed streams to all channels in C. We assume that the streams for the channels carry only messages of the right type.
We describe the behaviour of a component by a stream-processing function. It defines the relation between the input streams and output streams of a component that fulfills certain conditions with respect to their timing. A stream-processing function is represented by a set-valued function on valuations of the input channels by timed streams that yields the set of histories for the output channels

$$f : \vec{I} \to \mathcal{P}(\vec{O})$$

and that fulfills the timing property *time-guardedness*. This property axiomatises the time flow. It expresses that the set of possible output histories for the first $i+1$ time intervals only depends on the input histories for the first i time intervals. In other words, the processing of messages in a component takes at least one tick of time. For a precise formal definition of this property see [BS97].

2.2 Foundations of Object Orientation

Based on the theory given above, we have defined a set of concepts to give FOCUS an object-oriented flavor. This allows us to give a formal semantics to object-oriented modelling techniques, like UML [BRJ97] as we have done in [BHH+97].
For that purpose, we have defined a "system model" in [KRB96], that characterises our notion of object-oriented systems. Objects can be naturally viewed as components, as defined in the last section. Based on that, communication paths are defined using identifiers, where each object is associated with exactly one identifier (its identity).
In the system model, objects interact by means of *asynchronous message passing*. Asynchronous exchange of messages between the components of a system means that a message can be sent independently of the actual state of the receiver, as e.g. in C++ or Java. To model communication between objects we use the FOCUS basic data structure of streams and stream-processing functions.
Objects encapsulate data as well as processes. *Encapsulation of process* means that the exchange of a message does not (necessarily) imply the exchange of control: each object is regarded as a separate process. *Encapsulation of data* means that the state of an object is not directly visible to the environment, but can be accessed using explicit communication. The data part of the object defines its state. It is given in terms of typed attributes.
Objects are grouped into classes, that define the set of attributes of an object and its method interface (= message interface). This allows to model the behavior of the objects of each class c as stream-processing functions f_c mapping input histories to sets of output histories. As usual, classes are structured by an inheritance relation $\sqsubseteq$. We thus get a natural definition of inheritance of behavior if we demand that if a class inherits from another, its possible behaviors are a subset:

$$\forall c, d : Class.\ c \sqsubseteq d \ \Rightarrow\ f_c \subseteq f_d$$

In case of method extension, this constraint is adapted to an interface refinement constraint.
Dynamic and mobile features, such as creation of new instances and change of communication structures are also characterized as extension of Focus.

2.3 Refinement and Compositionality

Based on a first formal specification, the development of software and also of distributed systems is going through several development phases (or levels of abstraction). Through these phases the envisaged system or system component is described in an increasing amount of detail until a sufficiently detailed description or even an implementation of the system is obtained. The individual steps of such a process can be captured by appropriate notions of refinement. In a refinement step, parts or aspects of a system description are described more completely or more detailed. For this purpose, FOCUS offers a powerful compositional refinement concept as well as refinement calculi. On the semantic level, refinement is modeled by logical implication. The important refinement concepts are:

Behavioural Refinement: The aim of this refinement is the elimination of underspecification as it is needed e.g. for the specification of fault-tolerant behavior.

Interface Refinement: Here, the interface of a specification is refined by changing the number or types of the channels as it is needed for concretisation of messages or splitting communication connections between components.

Structural Refinement: This concept allows the development of the structure of the distributed system by refining components by networks of components.

2.4 A Formal System Development Process

FOCUS provides a general framework and a *methodology in the large* for the formal specification and stepwise top-down development of distributed reactive systems. The formal system development process consists of several phases of abstraction:
During the *Requirement Phase*, a first formalization of a given informal problem description is developed. Since the informal description is often not detailed enough, this first step of a system specification is often hard to develop. It is, however, essential for the formal system development, because it will be used as the basis for further development of specifications with growing degree of accuracy in the following phases. In this step, specifications can be formalized as either trace or functional specifications. The transition between these paradigms is formally sound and preserving correctness.
During the *Design Phase*, the essential part of the system development is carried out by developing the structure of a distributed system and refining it up to the intended level of granularity. These formal development steps are based on the specification determined in the requirement phase and their correctness will be shown relative to the first formalization. Because the formal development of a more detailed specification possibly uncovers mistakes or unprecise properties in earlier formalizations the top-down development is not linear but rather leads to re-specifications of some parts of earlier formalizations. Only the description of system properties in a mathematical and precise manner gives a system

developer the possibility to formally prove and refine system properties and descriptions. In this phase, in FOCUS the specifications are based on the denotational semantics which models component behaviour by stream-processing functions. For the development of the specifications during the design phase, paradigms like relational and functional specifications as well as several specification styles like Assumption/Commitment or equational specifications are defined. To increase its usability FOCUS is adapted for the use of various engineering oriented and practically used techniques and formalisms like tables or diagrams, see section 3. Due to the specific natures of these variants they can be used tailor-made for the solution of specific problems.
During the *Implementation Phase* the design specification is transformed into an implementation. This phase is subject of future work.

2.5 Further Work

Since the semantic foundations of FOCUS, including its development techniques are already explored in depth, the emphasis of further work lies on a better applicability of the methodology, especially for system developers less experienced in formal methods. For that purpose, additional wide-spread description techniques, (semi-) automatic and schematic proof support have to be offered. Several techniques for describing and specifiying systems (like tables, state or system diagrams, MSC-like event traces, the "Assumption/Commitment" style) were successfully integrated in the methodology. With AUTOFOCUS, tool support for system development is already available, giving future case studies a new quality by offering appropriate editors, consistency checks, code generation and even simulation. Current research activities concern the enhancement of FOCUS with methodical guidelines to ease the use of the mathematical formalism, the description techniques and the development methodology for non-specialists and to support solutions for specific application fields, like the modelling of operating systems concepts in [Spi98]. Case studies are an important and stimulating work for testing FOCUS in different application areas. FOCUS will be further improved, using the experience gained from the great number of case studies collected in [BFG+94] and [BBSS97] and future studies to come.

2.6 On the Direct Use of Formal Description Techniques

In the last sections we have sketched a mathematical framework for system specification. This allows developers to precisely describe structural and behavioural properties of the components and the composed system. As will be argued in section 3, one can hide the mathematics to developers through the use of graphical description techniques whose semantics is based on the formal framework. However, not everything can adequatly be expressed in diagrams. Especially, behavioural properties are difficult to express. Thus, for example, object-oriented specification methods typically use state transition diagrams to describe method acceptance in classes or collaboration diagrams to describe method calls between classes, but only programming language code to define the method bodies. Mathematical specification languages like FOCUS allow complete behaviour description in a much more declarative style. To be useful for practitioners, however, the notation must be simple and the specification language must be enhanced with guidelines for a systematic development of specifications. These guidelines are useful for developers

formulating properties of individual systems, as well as for method developers who need to state and verify properties of the (diagrammatic) description techniques on the basis of the formal semantics.
In the following we present an example of some guidelines to write down formal specifications in FOCUS. To make formal specification techniques and methods more acceptable it is essential that the developer is in the position to concentrate on his or her problem and not on the correctness of the formalization. In FOCUS, equations on stream-processing functions describe the mapping of patterns of input messages to patterns of output messages. [Spi98] proposes a special strategy to formulate the required behaviour as structured text. The translation of this text into a functional equation is supported by special schemes. In the following we show such a schema regarding a component C with one input channel In and one output channel Out, where messages of type *Integer* flow on these channels. We require that C computes the square of each input message and sends it on the output channel. For this input/output behaviour we give the following textual description:

> If the component C receives a message $X \in Integer$ on the input channel In, then C sends as reaction the square X^2 as output message on the output channel Out.

This structured text, which includes all information needed to specify the required behaviour, can be translated with the available schemes in the following functional equation (here f_C denotes the stream-processing function modelling the behaviour of the component C):

$$f_C(\{In \rightarrow X\} \circ s) \;=\; \{Out \rightarrow X^2\} \circ f_C(s)$$

3 Description Techniques

A description technique can be best characterized as a specialized language with the purpose of describing a particular view of the systems to be developed. With the FOCUS method, we have already been able to precisely define what our notion of a system is. It is then an important task to define an appropriate set of description techniques which allow developers to describe properties of systems.
In the first subsection, we will in general describe the notion of a description technique, how we treat them, and what the benefits of this treatment are.

3.1 Description Techniques, Notations and Semantics

A description technique serves the purpose of describing particular aspects (views) of a system. There exist a variety of graphical and textual description techniques, that allow to describe different aspects.
A description technique comes along with

- a concrete syntax (this is the concrete layout of all documents),
- an abstract syntax (without "syntactic sugar"),

- context conditions for wellformedness, and
- a semantics definition.

For a precisely defined description technique all four parts must be present. In case of textual notations, concrete and abstract grammars are common for the syntax, attributes on this grammar can be used for wellformedness conditions, and the semantics is usually defined as a mapping from the syntax in an appropriate semantic domain.
Similar techniques can be used for graphical notations. Each graphical notation basically defines a *language* of wellformed documents, which serves as the syntactic domain. As we want to use several description techniques for describing different aspects of the same systems, we need semantics definitions (mappings) that map the different syntactic domains onto the same semantic domain. This is the necessary basis to integrate the different description techniques during development. If we map different notations onto the same semantic domain, we (meaning the notation developer!) can compute context conditions between different notations, which ensure consistency of several views onto a system. Moreover, we can justify the correctnes of translations from one notation into another one, e.g. translating Message Sequence Charts into State Machines, or generating code. Last but not least, we can justify the correctnes of refinement calculi for the given descriptions.
There are other benefits of defining a precise semantics, e.g. the developer of the semantics gains a deeper understanding of the used notations. However, usually this formal semantics definition cannot be communicated to method users, only the (informal) interpretation of the insights can [FB97]. However, the most important bargain of precise semantics is the possibility to automate development steps.
As graphical techniques usually are not powerful enough to describe (or prove) every property of a system, it can be interesting to actually translate the documents of a notation into their "semantics" and use the power of the semantic formalism to specify further aspects.. In our case, e.g. different kinds of diagrams can be translated into formulas only using concepts of FOCUS.
In the following, we sketch the most important notations we have dealt with. We sketch the purpose of the notation and results, we have achieved on that notation.
We emphasize that it is important to also use explanations or other informal kinds of diagrams and text during development. A good method does not only deal with formal notations, but also allows the systematic treament of informal documents. The AUTOFOCUS tool uses a subset of the description techniques introduced below in variations that are tailored for the development of embedded systems (cf. Figure 2).

3.2 System Structure Diagrams (SSD)

System Structure Diagrams as used in AUTOFOCUS (Figure 2, upper middle) focus on the static structure of a system. They graphically exhibit the components of a system and their interconenctions. Such, they describe the glass box view of a FOCUS component and are therefore rather similar to ROOM charts [SGW94]. These diagrams focus more on the static part of a system and are not used in UML [BRJ97], where everything is assumed to be highly dynamic.

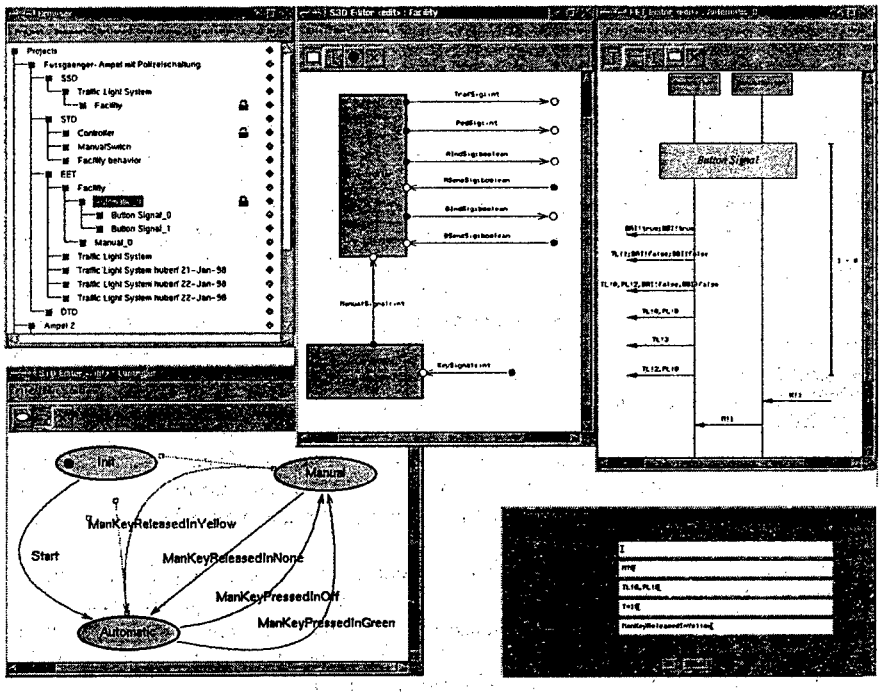


Figure 2: AUTOFOCUS Description Techniques: SSD, EET, and STD

Components may be hierachically decomposed. Therefore, for each non-elementary component such an SSD can be defined, leading to a hierachy of SSD documents describing a hierachical system structure.
If a system (or system component) exhibits dynamic properties, like changing the communication structure or creating/deleting components, the SSD can be used to describe structural snapshots or the static part of the structure. In an object-oriented flavor, an SSD defines a snapshot of data and communication paths between a set of objects.
As SSDs describe the architectural part of a system, there exists a refinement calculus for architectures, that allows to transform the internal structure of a component, e.g. by adding new components or changing communication paths, without affecting the external behavior of the component [PR97b, PR97c].

3.3 Class Diagrams (CD)

Class Diagrams are the most important object-oriented notation, and are therefore part of UML [BRJ97]. They are used to describe data aspects of a system as well as possible structure layouts. In contrast to System Structure Diagrams, that focus on the "instance level", Class Diagrams focus on the "type level". Each class may have several objects as

instances, each association represents links between appropriate objects.
Class Diagrams define a large class of possible structures. To further detail these structures, different kinds of invariants are added. E.g. associations have multiplicities, but in general, it is possible to add predicates defined in our Specification Language SL (see below).
Class Diagrams are also used to define the signature of a class and their state space. The signature consists of a set of method definitions, that also define the set of possible messages. The attributes define the state space.
In [BHH+97] we have argued about the semantics of Class Diagrams. Although Class Diagrams are a rather well understood technique, there are still open questions, e.g. on the treatment of aggregates.

3.4 Specification Languages (SL)

Not every aspect of a system can or should be described using graphic techniques. For example datatype definitions or additional constraints are best described using a textual notation. In UML e.g. OCL has been introduced for describing a certain type of constaints. However, as OCL does not allow to define data types or auxilary functions, and based on our experiences with algebraic specification techniques [BBB+85, BFG+93a], we have decided to define an own language for that purpose.
SL is an axiomatic specification language based on predicate logic, resembling Spectrum [BFG+93a, BFG+93b]. SL allows declarative definitions of properties. Particularly, SL is used for the definition of pre- and post-conditions of transitions and for the definition of state invariants not only in single objects but also between several objects in the Class Diagrams. In order to enable automatic testing of verification conditions, SL also incorporates concepts of functional programming, especially some taken from Gofer [Jon93]. The step from high-level descriptions towards executable code is facilitated, which in turn facilitates prototyping.
When restricting to the executable sublanguage, and furthermore to the datatype definitions, then an automatic translation into simulation code is possible.
We also have experimented with HOLCF [Reg94] as higher order logic as a property definition language, especially if used as a front end for the theorem prover Isabelle [Pau94].

3.5 Message Sequence Charts (MSC) and Extended Event Traces (EET)

Message Sequence Charts and Extended Event Traces are both used to describe the flow of communication within exemplary runs of a part of a system. Constituting a high level of abstraction, MSC are well suited to capture a system's requirements. Moreover, MSC can be used for and generated by simulation, respectively. We have developed different flavors of this technique. One focuses on synchronous message passing between different components [BHS96, BHKS97] and its semantics is primarily a set of traces. These are called Extended Event Traces and are used in AutoFocus (Figure 2, top right).
The other variant focuses on object-oriented systems and is more similar to MSC'96 [Int96]. Both variants are compared and argued about their semantics in [BGH+97a]. For the EET a set of operators was defined to combine EET sequentially, in parallel and

iterated. This allows not only to define exemplary behavior, but also complete sets of behaviors.
Currently, work is in progress to map EET into State Transition Diagrams.

3.6 State Transition Diagrams (STD)

In principle State Transition Diagrams describe the behavior of a component using the state of this component. But there are different abstractions and therefore flavors possible. E.g. STD can be used early in the development (analysis) and also in the design phase, when some kind of "lifecycle" of a component is modelled. During detailed design and also prototyping, pre- and postconditions of a certain form (e.g. executable) can be used to generate code.
We have explored and developed a whole variety of State Transition Diagrams, that allow to capture more than just one input or one output element on a transition. Usually a transition is attributed with a set of messages (sometimes restricted to one message) to be processed during the transition and a set of messages to be produced. There are timed and untimed variants, and there are variants incorporating pre- and postconditions on transitions [RK96, PR94, GKR96, GKRB96, GR95, Rum96, PR97a].
In the object-oriented flavor, State Transition Diagrams describe the lifecycle of objects. In STD, descriptions of state and behavior are combined. STD can be used at different levels of abstraction, that allow both the specification of an object's interface as well as the specification of individual methods. Refinement techniques enable not only inheritance of behaviour but also stepwise refinement of abstract STD [Rum96], resulting in an implementation.
A textual representation of State Transition Diagrams can be given using appropriate tables [Spi94, Bre97]. Hierachical variants of State Transition Diagrams are examined in [NRS96] and also used in AutoFocus (Figure 2, bottom left).
State Transition Diagrams are an extremely promising notation, as they allow on one hand to describe behavior, but on the other relate it to the state of a component. They allow to think in rather abstract terms of interaction sequences, but also can be used to describe a strategy of implementation (and therefore code generators). It is therefore worth to explore more precise variants of STD than the ones given in nowadays methods as UML.

3.7 Programming Language (PL)

The ultimate description technique is the target programming language. For object-oriented systems, Java [GJS96] is a rather interesting choice for an implementation language, as it exhibits a lot of desirable properties. It is not only a language with a set of consolidated and clear concepts, it also exhibits some notion of concurrency, which allows to implement the concurrency concepts of FOCUS. Hence, we have had a closer look on Java, e.g. selecting a suitable sublanguage which will be the target for our code generation from STD and MSC.
To include the programming language in a proper way into the formal development process, a step has been taken in [PR97a] towards a FOCUS based transitional semantics of conventional languages like Java.

3.8 Further Work

For some of the above described notations, we already have prototype tools, that allow us to edit and manipulate documents of that notation. Several others still need consolidation, as the process of finding not only a precise semantics for given notations, but adapting the notation in such a way, that it is comfortable to use and allows to express the desired properties, needs to do examples.
Currently also work to implement refinement calculi on Class Diagrams and State Transition Diagrams is in progress.

4 Methodical Ingredients

A software or system development method (cf. Section 5) covers a variety of different aspects. Description techniques, as introduced in Section 3, are only one of these aspects, yet probably the most "visible" one. However, a development method also contains a notion of a development *process*, a model, how developers proceed during the development of a system in order to produce the results (the documents, the specifications etc.) necessary for a complete and consistent system description that fulfills the requirements and ultimately results in the desired software product.
Such a process model usually operates on different levels of granularity, ranging from a coarse view down to very detailed, even atomic operations on specification elements or documents. The former will be treated in more detail in Section 5, while the latter are covered in this section.
Methodical steps can basically be partitioned in two disjoint sets of operations on specifications, operations that modify the *contents* of specifications, thus effectively yielding a different (possibly refined) description, and operations that change the (possibly informal) *status* of specifications, for instance from a draft status to a status "validated", indicating that certain properties of the specification have been found to be fulfilled in an informal process.
In the following sections, we give a set of examples for both kinds of steps that have been treated in our work.

4.1 Completeness and Consistency

Generally, a system specification, just like a program that is being written, is neither complete nor consistent most of the time within a development process. This is particularly the case in view-based systems development, which specifically aims at separating different aspects of a system description in different specification units (specifiation documents, for instance) that use a set of appropriate description techniques. From a methodical point of view, allowing for inconsistency and incompleteness during a development process is reasonable because enforcing them at any time restricts developers way too much in their freedom to specify systems. For instance, instead of concentrating on a certain aspect of a specification, developers, when changing parts thereof, would immediately have to update all other affected specification units that are possibly affected by such a change in order to maintain a consistent specification. Apart from diverting developers' attention from their current task, this is, especially with respect to completeness of specifications,

virtually impossible in practical development. Please note that the notion of consistency used here refers to the properties of the abstract syntax (the "meta-model") of the description techniques used to specify a system. Semantic aspects, such as, e.g., consistency of behavior with certain requirements are not treated in this context. This is quite similar to compilers for programming laguages which can ensure the "consistency" of a program, but not the correctness of the algorithm encoded in the program.
The AUTOFOCUS tool, which uses a view-based approach to specify distributed systems, offers such a mechanism to test specifications for completeness and consistency. System specification is based on a subset of the description techniques introduced in Section 3, namely, system structure diagrams, datatype definitions, state transition diagrams, and extended event traces. The view specifications covered by these techniques can be developed separately to a large extent. Only at specific points in the development process, for instance, when generating a prototype from a specification (cf. Section 4.2), some global conditions of consistency have to be fulfilled. Consequently, the consistency mechanism available in AUTOFOCUS is user-controlled and can be invoked at any time during development, allowing to select both an appropriate set of specifications to be checked and the (sub-) set of consistency conditions to be applied.

4.2 Validation of Specifications

In practical systems development, validation techniques, in contrast to formal verification techniques, are widely used today [BCR94] to get more confidence in the appropriate choices of the requirements. Verification techniques can only show the correctness of an implementation with respect to a specification. They will be treated in the next section. Validation techniques are the focus of this section. They cover a broad range of diverse techniques, such as

- reviews of specifications,
- systematic specification inspection,
- (usability) tests of software, or
- prototype generation and execution.

These techniques show different facets of validation. For instance, testing is applied usually for verifying that program code (that is, the ultimate target of a development process) fulfils certain required properties. Reviews and inspections techniques, in contrast to that, are applicable in virtually any stage in the development process to ensure consistency and certain correctness aspects on an informal level. Reviews, for instance, can be held upon requirements documents in the very early stages of a devlopment process, but as well on program code implemented by developers. Prototype generation for a system or parts thereof can be used once a specification has been developed that is sufficiently consistent and complete to validate the desired properties. Since a prototype, especially an executable prototype in the form of a program, virtually brings a system specification "into life" this kind of validation technique is especially relevant in communicating development results to customers. Prototyping has been successfully applied particluarly in areas like graphical user interfaces (GUI).

In software engineering, the usage of graphical formalisms that describe systems from a point of view rather close to an implementation is widespread. Examples for such techniques are statecharts [HPSS87] which are used in the STATEMATE tool [Ilo90] or state transition diagrams as used in the AUTOFOCUS tool, both of which can basically be regarded as a kind of graphical programming language. In such cases generating executable prototypes (or even generating final implementation code) is possible.
In the remainder of this section, we will take a brief look at such a prototyping environment, the AUTOFOCUS component SIMCENTER [HS97]. It is based on generating program code from a set of sufficiently detailed and consistent system specifications and on observing the behavior of that prototype program in its environment.
SIMCENTER works by generating Java program code from a specification of a distributed system, given in the AUTOFOCUS description techniques briefly outlined in Section 4.1. The generated program code, which is executed in SIMCENTER's runtime environment, is closely linked to a visualization component where the progress of the prototype execution can be monitored at the same level of description techniques as used to specify the system. A prerequisite for generating such an executable prototype, obviously, is that the specification is sufficiently complete and consistent in the sense outlined in Section 4.1. Nondeterminism, however, may be present in the behavioral aspects of the specification, they are currently resolved by selecting *one* possible behavior in the code generation process. This approach will be made more flexible from developers' point of view, for instance, by allowing them to select one of several nondeterministic behaviors during prototype execution.
As AUTOFOCUS' primary application domain are embedded systems, SIMCENTER allows to monitor the interactions of such a gerated prototype with its environment. In particular, developers are able to inject stimuli into the system and observe its reactions, both from its environment interface in a black box manner and from the internal perspective, as outlined above. Additionally, black box behavior of an embedded system prototype can be optionally observed and influenced from a user-definable, application domain-oriented environment view that can be attached to SIMCENTER via a standard communication interface. This allows developers a very customer-oriented presentation of the behavior of such a prototype and thus contributes to enhance communication between system developers and application domain experts.
For technical details about the process and the basics of code generation in SIMCENTER we refer the reader to [HS97], for an AUTOFOCUS development case study using SIMCENTER to validate certain correctness aspects of a specification of a simple embedded system, we refer to [HMS+98].

4.3 Verification Techniques

In contrast to informal validation, formal techniques allow developers to mathematically prove that a system specification fulfills certain requirements. As a prerequisite, both the requirements and the specifications need to be formulated in notations that have a common mathematical basis, thus allowing formal proofs to be conducted.
Our goal is to integrate formal techniques as seamless as possible with some of the description techniques introduced in Section 3. Within the AUTOFOCUS project two categories of verification tools are currently under consideration for an integration with graphical

formalisms. First, interactive theorem provers like *Isabelle* [Pau94] in conjunction with HOLCF [Reg94] could be used to interactively prove properties of a specification. For that purpose, graphical specifications would have to be transformed into the HOLCF notation, and developers would have to conduct their proofs on the HOLCF level of description. Obviously, this approach is not very intuitive because it forces developers that are used to graphical notations to use a mathematical formalism to conduct proofs.
Thus, the second category of tools, automated verification tools like model checkers seem to be more suitable for a seamless integration. Currently, a prototype for the integration of the μ-cke model checker [Bie97] into AUTOFOCUS is being implemented. It will be able to check whether a concrete system specification, given by a component network and the corresponding behavioral descriptions, exposes a refinement of the behavior of a given, more abstract specification.

4.4 Transformations

Transformations are methodical steps that effectively change a system description. Thus, each action by developers that add or change specification elements results in a different system description. Whether such modifications to specifications preserve certain properties of a specification that have been established before, is not a priori clear and has thus again to be validated (or verified, in case of a formal development process). For that reason, it is desirable, and feasible as well, to have a class of methodical steps that allow developers to change specifications in a way that previously estblished properties will still hold after the modifications. Providing such property-preserving modification steps for a set of object-oriented description techniques is one of the main goals of the SYSLAB project. Such property-preserving transformations are defined on the level of the description techniques and provided to developers in the form of a syntactical refinement calculus that will be integrated in the toolset that is currently being developed within SYSLAB. These transformation rules are formally proven to be property-preserving by the method developers and thus enable system developers to perform transformations on specifications on the syntactical level without having to re-establish the validity of previously valid properties. Currently, such transformation calculi exist for state transition diagrams [Rum96] and for system structure diagrams [PR97b, PR97c] and are being integrated into the SYSLAB toolset. If developers choose not to use transformations provided by the refinement calculus, but to make arbitrary, manual modifications to their specifications they have to explicitly re-establish the necessary properties again.

4.5 Further Work

In the context of methodical development steps, tool-based active developer support is a major area of work in the near future. One aspect consists in guiding developers through the development process, offering them possible development steps that can be or need to be performed in order to develop a system.
Another important aspect consists in tracing the development steps applied to specifications and their effects on other specifications. This pertains both to syntactic consistency and completeness of the specifications and to possibly invalidated semantic properties that need to be re-established after development steps.

5 A Model-Based Software Development Process

Up to now we have looked at formal modeling techniques, tool-support for model development and analysis based on an integrating formal basis, and a formal development process. The modeling techniques mentioned above aim at the description of the software system on various levels of granularity. We show in the following, that they can naturally be complemented with a set of description techniques for the software system context and the informal problem description. We will sketch a framework for a model-based development process. This framework is made up of three main ingredients [Jac95]:

- the distinction between the world, the machine and their interface and the explicit system models of all three of them,
- the distinction between the external view, the internal analysis view and the (distributed) design view of each system, and
- a careful deployment of formality.

These three issues will be discussed in the following subsections. Depending on the application domain and the project context this framework needs to be instantiated. We sketch an example process for information system development at the end of this section.

5.1 The World, the Machine and their Interface

The distinction between *the world and the machine* is due to Jackson [Jac95]. The problem to be solved by a software system is in the world, the machine constitutes the solution we construct. Phenomena shared by the world and the machine make up the *interface*. Descriptions produced during software development must be clearly associated to one of the these three domains. This is especially difficult for requirement documents, which typically contain references to the world, namely the effects to be achieved by the software system, to the interface, namely the system services, and the machine. In particular, it is not possible to describe the system services precisely without a clear understanding of the relevant phenomena of the world. Therefore software engineering methods - formal or pragmatic - typically start with informal descriptions of the issues in the world relevant to the software system. These are then transformed into so-called analysis models. The modeling techniques used for these models are the same as the ones used for the description of the machine. Object-oriented methods like OMT [RBP+91] or OOSE [Jac92] use object models, structured methods like SSADM [DCC92] use dataflow models. This is reasonable, because the world and the machine can both be viewed as systems allowing therefore for the same modeling techniques. However, there are semantical differences: in object models of the software systems associations represent references directly implementable in the programming language. Associations between objects in the world represent invariant relationships which typically manifest themselves as natural phenomena (e.g. a person has mother and father) or as social or legal processes (e.g. a book has an author). Also, the purpose of the models of the world and the machine is quite distinct. Models of the world capture understanding of important phenomena (expressing the indicative mood according to [Jac95]), while models of the software system capture requirements to be

realized by the software system or document the running system (expressing the optative or the indicative mood, respectively).
To make these distinctions explicit, we therefore distinguish three categories of models:

models of the world They model the context of the software system, e.g. a railway system or a lift to be controlled by the software system, or an production company whose engineers are supported by software systems. In particular, it is important to model the processes which the software system is involved in.

models of the interface They model the phenomena shared between the world and the machine. In particular, it is important to model the interaction between the software system and its external partners. The later may be humans or machines.

models of the machine They model the internals of the software system, namely the internal components (e.g. objects, subsystems) and how they render the system services.

5.2 The External View, the Internal View and the Design View

The world, the interface and the machine constitute systems. They all consist of actors, communicating with each other, and executing activities making use of their (data) ressources. Figure 3 collects elements of the three different systems in case of a railway control system:

	actors	**data**	**activities**
world	trains, passengers, conductor	timetable, position	passengers enter and get off the train, train stops
interface	train personnel, software system	signals	signaling, to switch the points
machine	objects, operating system processes	attributes	assignment, method call

Figure 3: The world, the interface and the machine as systems

Software development methods traditionally either focus on the activities and their data flow (structured methods) or on the actors and their communication (object oriented methods). We claim that both views are important during system development, and that a third view has to be explicit: the *external* view. The external view describes the services to be delivered by the system. The activities describe steps to achieve the required services. We call activities and their data the *internal analysis view*, because at this level one experiments with different ways of achieving the services without regard for the actors. The actors constitute the *distributed design view*. Activities and data are encapsulated within actors, such that data flow between activities has to be realized through communication. As exemplified by object oriented designs, an actor-oriented structure allows better reusability and extensibility of designs than activity-structured designs.

Each of these views can be applied to the world, the interface and the machine. To understand the purpose of the context of the software system, it is usually helpful to describe the services of this context. In the case of the railway control the services are the transport services offered by the trains at particular locations and at particular times. In order to adequately understand the services, the activities and data of the world have to be modelled quite extensively. The actor structure of the world is very often changed by introduction of the software system, since often human labour is replaced. Also, it is very often subject to a lot of political decisions.
The services of the interface are the work processes or technical processes to be supported by the software system. Jacobsen [Jac92] has coined the term *use case* for this. Very often there is a close correspondence between machine and interface services, the latter being a high-level view of the former. The analysis and the design view of the interface are heavily intertwinned. In the interface the actors are mostly given (humans and technical systems), but there is a choice of how to distribute the activities between the machine and the external partners.
The services of the machine are determined by the design of the interface. Typically, the service view and the analysis view of the machine is heavily intertwinned, because the services cannot be described without resorting to the data of the software system. Often, also some parts of the design view are fixed, because the machine has to fit into an already existing landscape of software systems. Thus, for example, one actor maybe a particular database, other actors may be given by a library of classes for a particular application domain.

5.3 An Example Process

The discussions above can be captured in the following proposal for a development process for informations systems covering the external, internal and design views for the world, the interface and the machine. The formal system descriptions and development steps discussed in the previous sections are typically only used for the machine view. Only if the effects of the software system in the world are critical (e.g. chemical processes), formalization of the world and interface models will be worthwile.
Figure 4 lists the models for developing a software system design.
This process is influenced on the one hand by SSADM [DCC92], especially regarding the transition from the machine services to the machine analysis view. It has similarities to OOSE in the use of use cases for the external view of the interface. The transition from the machine analysis view to the machine design using exemplary communication flow descriptions like EET is borrowed from FUSION [CAB+94].
Of course, this process is only a framework to be instantiated for different application domains and projects. The interface models have to be quite detailed in case of human-computer interaction with a new technology [Suc95]. The world models have to be quite detailed in case of a new or critical application domain. Models of the software system should allow for a systematic transition to code using the development steps described in section 4.

View	World	Interface	Machine
service specification	(textual) description of the enterprise services	use case model listing the user tasks	system services (specified in terms of their input/output and/or the data changes)
data and activity analysis	glossary, application domain processes	work processes or technical processes	data model described as ERD or CD, data changes described by STD
actor and communication design	(textual) description of the responsibility (in terms of data and activities) of the departments	(textual) description of user roles and technical system partners, allocation of data and activities to software system	description of the component-oriented design by SSD, CD, STD, EET

Figure 4: Products of a model-based software development process

6 Conclusion

The paper has discussed the issues of using formally founded description techniques for system and software engineering. We have shown that formal methods like FOCUS provide a rich basis for textual and graphical system descriptions, as well as the basic methodical steps for system development. This formal basis allows for an integrated view on the wealth of description techniques found in the literature. Equally important for the system developer are the methodical elements based on the formal semantics like consistency checks and transformations. For real world application, this formal development process has to be embedded into a process of application domain (world) and usage (interface) understanding and description. From our experience, each of these issues is worth its own project. Our projects have demonstrated that it is possible to resolve each of these issues on its own, restricted to a particular application domain. The challenge is now to connect all of this together and to transfer it to new application domains. This can only be achieved by a widespread use of these techniques in university and industry.

Acknowledgments

We like to thank all the people who have contributed to the work presented in this paper, especially those involved in the projects FOCUS, AUTOFOCUS, FORSOFT and SYSLAB.

References

[BBB+85] F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing und

H. Wössner. *The Munich Project CIP, Vol 1: The Wide Spectrum Language CIP-L.* LNCS 183. Springer-Verlag, 1985.

[BBSS97] M. Broy, M. Breitling, B. Schätz und K. Spies. Summary of Case Studies in FOCUS – Part II. SFB-Bericht 342/24/97 A, Technische Universität München, September 1997.

[BCR94] V.R. Basili, G. Caldiera und H.-D. Rombach. Goal Question Metric Paradigm. In J.J. Marciniak, Hrsg., *Encyclopedia of Software Engineering*, Seiten 528–532. John Wiley & Sons, 1994.

[BDD+93] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner und R. Weber. The Design of Distributed Systems – An Introduction to FOCUS. SFB-Bericht Nr. 342/2-2/92 A, Technische Universität München, January 1993.

[BFG+93a] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, O. Slotosch und K. Stølen. The Requirement and Design Specification Language SPECTRUM, An Informal Introduction, Version 1.0, Part 1. Technischer Bericht TUM-I9312, Technische Universität München, 1993.

[BFG+93b] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, O. Slotosch und K. Stølen. The Requirement and Design Specification Language SPECTRUM, An Informal Introduction, Version 1.0, Part 2. Technischer Bericht TUM-I9312, Technische Universität München, 1993.

[BFG+94] M. Broy, M. Fuchs, T. F. Gritzner, B. Schätz, K. Spies und K. Stølen. Summary of Case Studies in FOCUS — a Design Method for Distributed Systems. SFB-Bericht 342/13/94 A, Technische Universität München, June 1994.

[BGH+97a] R. Breu, R. Grosu, Ch. Hofmann, F. Huber, I. Krüger, B. Rumpe, M. Schmidt und W. Schwerin. Exemplary and Complete Object Interaction Descriptions. In H. Kilov, B. Rumpe und I. Simmonds, Hrsg., *Proceedings OOPSLA'97 Workshop on Object-oriented Behavioral Semantics.* TUM-I9737, 1997.

[BGH+97b] R. Breu, R. Grosu, F. Huber, B. Rumpe und W. Schwerin. Towards a Precise Semantics for Object-Oriented Modeling Techniques. In J. Bosch und S. Mitchell, Hrsg., *Object-Oriented Technology, ECOOP'97 Workshop Reader.* Springer Verlag, LNCS 1357, 1997.

[BHH+97] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe und V. Thurner. Towards a Formalization of the Unified Modeling Language. In *ECOOP, LNCS 1241*, Seiten 344–366, 1997.

[BHKS97] M. Broy, C. Hofmann, I. Krueger und M. Schmidt. Using Extended Event Traces to Describe Communication in Software Architectures. In *Proceedings APSEC'97 and ICSC'97,*. IEEE Computer Society, 1997.

[BHS96] M. Broy, H. Hußmann und B. Schätz. Formal Development of Consistent System Specifications. In M.-C. Gaudel und J. Woodcock, Hrsg., *FME'96:*

Industrial Benefit and Advances in Formal Methods, LNCS 1051, Seiten 248-267. Springer, 1996.

[Bie97] A. Biere. *Effiziente Modellprüfung des μ-Kalküls mit binären Entscheidungsdiagrammen.* Dissertation, Universität Karlsruhe, 1997.

[BMS96] M. Broy, S. Merz und K. Spies, Hrsg. *Formal Systems Specification - The RPC-Memory Specification Case Study, LNCS 1169.* Springer, 1996.

[Bre97] M. Breitling. Formalizing and Verifying TIMEWARP with FOCUS. SFB-Bericht 342/27/97 A, Technische Universität München, 1997.

[BRJ97] G. Booch, J. Rumbaugh und I. Jacobson. The Unified Modeling Language for Object-Oriented Development, Version 1.1, 1997.

[BS97] M. Broy und K. Stølen. FOCUS on System Development - A Method for the Development of Interactive Systems, 1997. Manuskript.

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes und P. Jeremaes. *Object-Oriented Development - The FUSION Method.* Prentice Hall, 1994.

[CGR93] D. Craigen, S. Gerhart und T. Ralston. Formal Methods Reality Check: Industrial Usage. In *FME, LNCS 670*, Seiten 250–267. Springer, 1993.

[DCC92] E. Downs, P. Clare und I. Coe. *Structured Systems Analysis and Design Method: Application and Context.* Prentice-Hall, 1992.

[DHP+98] B. Deifel, U. Hinkel, B. Paech, P. Scholz und V. Thurner. Die Praxis der Softwareentwicklung: Eine Erhebung. submitted to publication, 1998.

[FB97] R. B. France und J.-M. Bruel. Integrated Informal Object-Oriented and Formal Modeling Techniques. In H. Kilov und B. Rumpe, Hrsg., *Proceedings ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques.* Technische Universität München, TUM-I9725, 1997.

[GJS96] J. Gosling, B. Joy und G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[GKR96] R. Grosu, C. Klein und B. Rumpe. Enhancing the SysLab System Model with State. TUM-I 9631, Technische Universität München, 1996.

[GKRB96] R. Grosu, C. Klein, B. Rumpe und M. Broy. State Transition Diagrams. TUM-I 9630, Technische Universität München, 1996.

[GR95] R. Grosu und B. Rumpe. Concurrent Timed Port Automata. TUM-I 9533, Technische Universität München, 1995.

[HMS+98] F. Huber, S. Molterer, B. Schätz, O. Slotosch und A. Vilbig. Traffic Lights - An AutoFocus Case Study. In *International Conference on Application of Concurrency to System Design.* IEEE CS Press, 1998. to appear.

[HPSS87] D. Harel, A. Pnueli, J.P. Schmidt und R. Sherman. On the Formal Semantics of Statecharts. Proceedings on the Symposium on Logic in Computer Science, Seiten 54 – 64, 1987.

[HS97] F. Huber und B. Schätz. Rapid Prototyping with AutoFocus. In A. Wolisz, I. Schieferdecker und A. Rennoch, Hrsg., *Formale Beschreibungstechniken für verteilte Systeme, GI/ITG Fachgespräch 1997, pp. 343-352.* GMD Verlag (St. Augustin), 1997.

[HSS96] F. Huber, B. Schätz und K. Spies. AutoFocus - Ein Werkzeugkonzept zur Beschreibung verteilter Systeme. In U. Herzog und H. Hermanns, Hrsg., *Formale Beschreibungstechniken für verteilte Systeme*, Seiten 165–174. Universität Erlangen-Nürnberg, 1996. Erschienen in: Arbeitsberichte des Insituts für mathematische Maschinen und Datenverarbeitung, Bd.29, Nr. 9.

[Huß97] H. Hußmann. *Formal Foundations for Software Engineering Methods, LNCS 1322.* Springer, 1997.

[Ilo90] i-Logix Inc., 22 Third Avenue, Burlington, Mass. 01803, U.S.A. *Languages of Statemate*, 1990.

[Int96] International Telecommunication Union, Geneva. *Message Sequence Charts*, 1996. ITU-T Recommendation Z.120.

[Jac92] I. Jacobson. *Object-Oriented Software Engineering.* Addison-Wesley, 1992.

[Jac95] M. Jackson. The World and the Machine. In *ICSE-17*, Seiten 283–294, 1995.

[Jon93] M. P. Jones. *An Introduction to Gofer*, Manual, 1993.

[KRB96] C. Klein, B. Rumpe und M. Broy. A Stream-based Mathematical Model for Distributed Information Processing Systems - SysLab system model - . In E. Naijm und J.-B. Stefani, Hrsg., *FMOODS'96 Formal Methods for Open Object-based Distributed Systems*, Seiten 323–338. ENST France Telecom, 1996.

[NRS96] D. Nazareth, F. Regensburger und P. Scholz. Mini-Statecharts: A Lean Version of Statecharts. Technischer Bericht TUM-I9610, Technische Univerität München, 1996.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover, LNCS 828.* Springer-Verlag, 1994.

[PR94] B. Paech und B. Rumpe. A New Concept of Refinement Used or Behaviour Modelling with Automata. In *FME'94, Formal Methods Europe, Symposium '94*, LNCS 873. Springer-Verlag, Berlin, Oktober 1994.

[PR97a] B. Paech und B. Rumpe. State Based Service Description. In J. Derrick, Hrsg., *Formal Methods for Open Object-based Distributed Systems.* Chapman-Hall, 1997.

[PR97b] J. Philipps und B. Rumpe. Refinement of Information Flow Architectures. In M. Hinchey, Hrsg., *ICFEM'97 Proceedings, Hiroshima. Japan.* IEEE CS Press, 1997.

[PR97c] J. Philipps und B. Rumpe. Stepwise Refinement of Data Flow Architectures. In M. Broy, E. Denert, K. Renzel und M. Schmidt, Hrsg., *Software Architectures and Design Patterns in Business Applications.* Technische Universität München, TUM-I9746, 1997.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy und W. Lorensen. *Object-oriented Modeling and Design.* Prentice-Hall, 1991.

[Reg94] F. Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF.* Dissertation, Technische Universität München, 1994.

[RK96] B. Rumpe und C. Klein. *Automata Describing Object Behavior*, Seiten 265-287. Kluwer Academic Publishers, Norwell, Massachusetts, 1996.

[Rum96] B. Rumpe. *Formal Method for the Development of Distributed Object-Oriented Systems (in German).* Herbert Utz Verlag Wissenschaft, 1996. PhD thesis, Technische Universität München.

[SGW94] B. Selic, G. Gulkeson und P. Ward. *Real-Time Object-Oriented Modeling.* John Wiley and Sons, 1994.

[Spi94] K. Spies. Funktionale Spezifikation eines Kommunikationsprotokolls. SFB-Bericht 342/08/94 A, Technische Universität München, May 1994.

[Spi98] K. Spies. *Eine Methode zur formalen Modellierung von Betriebssystemkonzepten.* Dissertation, Technische Universität München, 1998.

[Suc95] L. Suchman(ed.). Special Issue on Representations of Work. *CACM*, 38(9), 1995.

[Tur93] K.J. Turner(ed.). *Using Formal Description Techniques - an Introduction to ESTELLE, LOTOS and SDL.* John Wiley & Sons, 1993.

[Wor92] J.B. Wordsworth. *Software Development with Z.* Addison-Wesley, 1992.

Formal Methods and Industrial-Strength Computer Networks

J N Reed
Oxford Brookes University, Oxford, UK
jnreed@brookes.ac.uk

Abstract

Two case studies involving the application of formal methods to industrial-strength computer networks are described. In both case studies, the formal method (CSP/FDR) was thought sufficiently mature for these applications. However in both cases, for the formal method to be effective it was necessary to develop techniques requiring expert knowledge in the theory underpinning the formal method. These examples illustrate that there remain significant technical challenges to effective use of formal methods, which come to light only through large-scale applications.

Keywords: Formal Methods, Network Protocols, CSP, FDR.

1. Introduction

There are many varieties of formal methods, a term referring to the application of mathematics and mathematically derived techniques to the specification and development of program code and hardware. They all have the same purpose: improving the quality and reliability of computer software and hardware.

There are also numerous applications of formal methods. The overwhelming majority of these applications have been conducted by specialists in the formal techniques rather than by specialists in the application domain. I will describe two industrial-strength case studies, which help illustrate why application specialists do not yet effectively use formal methods. In both cases the formal method had previously been thought sufficiently mature for technology transfer; but disappointingly the method was found to have an inadequate match of existing techniques to the particular application domain. Happily in both cases the theory underlying the formal method was further investigated and focussed on the problem at hand, in order to provide suitable techniques; and the case-studies were successfully completed. However, developing suitable techniques which rendered the application problems tractable required considerable knowledge of the finer points of theory underpinning the formal method - of the sort it is not realistic to expect

practitioners to possess.

The first case study briefly described below involves a specification and verification of a signalling protocol for a realistic pots, "plain old telephone system" [KR93]. The work was done in the late 1980's as a part of the REX project, which was an ESPRIT collaboration among academia and industry. We constructed a high-level specification of the system using Timed CSP (TCSP) [RR86] and a refinement also using TCSP, and proved that the refinement met the specification. The problem was one of "formal clutter" - an excess of formal expressions at the top-most abstract level so as to render the problem of proof intractable by hand, or otherwise. We solved this problem by developing proof conditions whereby constraints which were relied on by one component were guaranteed by another component. This "rely and guarantee" technique significantly reduced the size of the specifications which had to be constructed and manipulated. However it was important that we establish that the these proof rules were theoretically sound, in particular, that they did not produce circular reasoning.

The second case study described below is part of ongoing research involving application of an automated property checker, FDR [FDR94], to modern high-speed, multiservice networks. FDR is a finite-state model checker for the process-algebraic language of CSP. Modern multiservice networks such as the Internet typically use protocols designed to operate with arbitrary numbers of interacting components. A problem in employing finite-state model checkers for these protocols is that the model checkers can not directly handle end-to-end properties of arbitrary but unbounded numbers of subcomponents. In order to use FDR for the Internet reservation protocol, we first had to develop an inductive approach for establishing properties of interest, including deadlock and livelock freedom, for such end-to-end protocols.

1 CSP and FDR

CSP [Hoa85] models a system as a *process* which interacts with its environment by means of atomic *events*. Communication is synchronous; that is, an event takes place precisely when both the process and environment agree on its occurrence. CSP comprises a process-algebraic programming language together with a related series of semantic models capturing different aspects of behaviour. A powerful notion of refinement intuitively captures the idea that one system implements another. Mechanical support for refinement checking is provided by Formal Systems' FDR refinement checker, which

also checks for system properties such as deadlock or livelock.

The simplest semantic model identifies a process as the sequences of events, or *traces* it can perform. We refer to such sequences as *behaviours*. More sophisticated models introduce additional information to behaviours which can be used to determine liveness properties of processes.

We say that a process P is a refinement of process Q, written $Q \sqsubseteq P$, if any possible behaviour of P is also a possible behaviour of Q. Intuitively, suppose S (for "specification") is a process for which all behaviours are in some sense acceptable. If P refines S, then the same acceptability must apply to all behaviours of P. S can represent an idealised model of a system's behaviour, or an abstract property corresponding to a correctness constraint, such as deadlock freedom.

The theory of refinement in CSP allows a wide range of correctness conditions to be encoded as refinement checks between processes. FDR performs a check by invoking a normalisation procedure for the specification process, which represents the specification in a form where the implementation can be validated against it by simple model-checking techniques. When a refinement check fails, FDR provides the means to explore the way the error arose. The system provides the user with a description of the state of the implementation (and its subprocesses) at the point where the error was detected, as well as the sequence of events that lead to the error. The definitive sourcebook for CSP/FDR can now be found in [Ros97].

Unlike most packages of this type, FDR was specifically developed by Formal Systems for industrial applications, in the first instance at Inmos where it is used to develop and verify communications hardware (in the T9000 Transputer and the C104 routing chip). Existing applications include VLSI design, protocol development and implementation, control, signalling, fault-tolerant systems and security. Although the underlying semantic models for FDR do not specifically address time (in contrast to Timed CSP formalism [RR86, TCSP92, KR93]), work has been carried out modelling discrete time with FDR [Sid93, Ros97, R98]. A class of embedded real-time scheduler implementations [Jac96] is analysed with FDR by extracting numerical information from refinement checks to show not only that a timing requirement is satisfied, but also to determine the margin by which it is met.

2. A POTS Signalling Protocol

In [KR93] we described a Timed CSP specification of a telephone exchange, together with a decomposition into a design also described using TCSP. We provided selected proofs establishing that the design satisfies the specification. The work was based on a large specification given in SDL by a telecommunications software company which formed a major part of the REX Esprit Project. The specification was not concerned with billing and data-transfer, rather with safety and liveness properties of the signalling phase of a "Plain Old Telephone Service". It treated awkward race conditions such as a caller replacing just as the callee telephone is about to ring.

We found that for this relatively large, complex application there was a tension between writing strong specifications (in order to achieve desired behaviour, and reduce the formal clutter and state explosion for both the specification and further refinements) and keeping the specification weak enough that it could be implemented. This tension does not reveal itself in the small sized examples underlying the intuition and test beds for much of the theoretical work on formal methods.

Strong specifications for individual subcomponents allow us to prove many properties about the composite system made up the components. However, it may be impossible or impractical to implement a component which satisfies a desired specification in every possible environment. For these cases it is desirable to relax (or weaken) component specifications but not so much that it becomes impossible to prove the composite system correct. We developed a *Rely and Guarantee* method for CSP which controls this relaxation by explicitly describing a component's intended environment. An added benefit is that the method can greatly reduced the "formal clutter" problem.

An example of a proof rule (stated intuitively for two components) for safety properties is the following : if S_p and S_q are initially true (true for the empty trace) and the events which cause S_p and S_q to be untrue are mutually exclusive, then

$$\frac{\begin{array}{lll} P & \text{sat} & S_p \Rightarrow S_q \\ Q & \text{sat} & S_q \Rightarrow S_q \end{array}}{P||Q \text{ sat } S_p \wedge S_q}$$

We use the above proof rule when we want to design a system to meet safety properties S_p and S_q, but we do not want to implement S_p and S_q unconditionally for P and Q respectively. Rather the implementor of P can assume S_q while implementing S_p, and the implementor of Q can assume

S_p while implementing S_q, that is the pair of weaker constraints $S_p \Rightarrow S_q$ and $S_q \Rightarrow S_q$ with processes P and Q. There is an apparent circularity here in that if Q, fails (i.e., fails to satisfy S_q), then P is no longer constrained and so may fail too, thus justifying Q's failure. The side conditions achieve a resolution by ensuring that P and Q are initially correct, and cannot go wrong simultaneously. Analogous but more complex rules are formulated for liveness properties.

The side conditions are automatically true (hence requiring no additional burden of proof) if there are no safety assumptions (constraints) placed on inputs. This makes intuitive sense because otherwise we would require the implementor to filter inputs according to value - something not always appropriate or efficient to do. For systems such as signalling protocols, components must sensibly deal with whatever inputs they are given, so these side conditions are automatically met.

We developed an architectural structure for organising and manipulating the specifications which substantially reduce the sheer volume of formal objects to be handled using this technique. The rely and guarantee parts of the specification could be collected together to form interface specifications, making for a high-level of organisation with a minimum of effort. This reduction in both effort and formal clutter proved the key to effectively formalising the POTS protocol.

The details of this rely and guarantee technique can be found in [KR93]. The observation of interest here is that for the TCSP formal method to be effective for this industrial-strength application, a new technique (the rely and guarantee) first had to be developed. There were two aspects of this new technique:

- The proof rules which had to be shown sound using the underlying theory of TCSP, and
- The architectural organisational/structuring conventions which substantially reduced the volume of detail.

The structuring conventions were somewhat application related and it might be hoped that application specialists could have developed such enabling techniques "in house", where necessary. However, establishing that the proof rules were sound required special expertise which we should not realistically expect application specialist to possess.

3. An Internet Reservation Protocol

Our second case study of interest involves certain aspects of the Internet RSVP protocol. We (J Reed, D Jackson, B Deianov and G Reed) used CSP/FDR to model and automatically check properties satisfied by the end-to-end protocol [R98].

RSVP is a protocol for multicast resource reservation intended for IP based networks. The protocol addresses those requirements associated with a new generation of applications, such as remote video, multimedia conferencing, and virtual reality, which are sensitive to the quality of service provided by the network. These applications depend on certain levels of resource (bandwidth, buffer space, etc.) allocation in order to operate acceptably. The RSVP approach is to create and maintain resource reservations along each link of a previously determined multicast route, with receivers initiating the resource requests. Thus it is analogous to a signalling phase prior to packet/cell transmission (such as found in ATM networks) during which virtual channels with associated resource assignments are put in place. The multicast may consist of several senders and several receivers.

The full technical specification for RSVP as given by its developers appears as a working document of the Internet Engineering Task Force [BZB96]. The protocol assumes a multicast route, which may consist of multiple senders and receivers. RSVP messages carrying reservation requests originate at receivers and are passed upstream towards the senders. Along the way if any node rejects the reservation, a RSVP reject message is sent back to the receiver and the reservation message discarded; otherwise the reservation message is propagated as far as the closest point along the way to the sender where a reservation level greater than or equal to it has been made. Thus reservations become "merged" as they travel upstream; a node forwards upstream only the "maximum" reservation request.

Receivers can request confirmation messages to indicate that the request was (probably) successful. A successful reservation propagates upstream until it reaches a node where there is a (pending) smaller or equal request; the arriving request is then merged with the reservation in place and a confirmation message sent back to the receiver. Thus the receipt of this confirmation is taken to be a (high-probability) indication rather than a guarantee of a successful reservation. There is no easy way for a receiver to determine if the reservation is ultimately successful. Enhancements involve control packets travelling downstream following data paths, which contain pertinent information to predict the result.

Several interesting aspects emerge from the intuitive description of the RSVP protocol. The protocol is defined for *arbitrary* routing graphs consisting of several senders and receivers. Confirmations sent by intermediate nodes to receivers are ultimately valid only for the receiver making the largest request; i.e., a requester may receive a confirmation although subsequently the end-to-end reservations fails because of further upstream denial. Clearly we are dealing with end-to-end properties inherently involving arbitrary configurations of intermediate nodes. Global views involving intermediate nodes, (e.g., a successful reservation propagates upstream until it reaches a node where there is a (pending) smaller or equal request) present serious problems indeed for building models consisting of predetermined sets of components.

Previous CSP/FDR network applications primarily centre on protocols, but these applications do not specifically address arbitrary network topologies. There are numerous examples of formalisations of layered protocols using a variety of techniques and approaches, including Ethernet - CSMA/CD (in non-automated TCSP [Dav91]) (in non-automated algebraic-temporal logic [Jma95]), TCP (in non-automated CSP [GJ94]), DSS1 / ISDN SS7 gateway (in LOTOS [LY93]), ISDN Layer 3 (in LOTOS [NM90]), ISDN Link Access Protocol (in Estelle [GPB91]), ATM signalling (in TLT, a temporal logic/UNITY formalism [BC95]). Shankar [Shan] uses an induction scheme for model-checking in PVS for Peterson's shared memory algorithm for mutual exclusion, but to our knowledge nothing in the literature specifically address the problem of modeling arbitrary network configurations such as the Internet with finite-state model checkers.

An Induction Scheme

We approached this problem by developing an induction scheme which lets us infer properties about arbitrary (but finite) collections of nodes from a small number of proofs about fixed numbers of nodes. For example, we might wish to establish deadlock or livelock freedom for an end-to-end protocol which operates with an arbitrary number of intermediate network nodes. We would therefore want to express models and properties in a topology independent manner. To achieve this, we base our specification on a fixed number of single network nodes together with their immediate neighbours, and inductively establish the property for arbitrary chains of such nodes.

Suppose we can characterise the interface which a sender or routing node presents to the next node downstream by a property P. Considering a single node (or partial node where splitting has been used to avoid cycles), if we can

demonstrate that under the assumption that all incoming interfaces satisfy P then so do all outgoing interfaces, we have established an inductive step which allows arbitrary acyclic graphs to be built up, always presenting an interface satisfying P to the nodes downstream. The essential base condition, of course, is that an individual data source meets P. The symmetric case starting with a property of a receiving node and building back towards a source is equally sound. A rigorous presentation of this inductive technique is given in [CR, Cre].

The essence of the method applied to the reservation protocol is to check an assertion effectively stating that if "upstream" channels of a module satisfy property P, then the "downstream" ones do likewise. Figure 1 illustrates the FDR mechanism to do this: assert that the parallel composition of a given module with a property satisfying P, with all upstream channels and all but one downstream channels appropriately hidden (made internal), refines P itself.

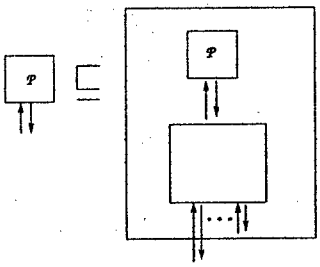


Figure 1: Simple Induction Scheme

The power of this modelling strategy depends on the ability to reduce a collection of arbitrary n processes to a fixed number of processes, which can then be mechanically model-checked. The reduction is possible only if an arbitrary process is defined recursively. In our examples, the state space is kept finite by limiting the resource set to a fixed number, and bounded by reducing an arbitrary chain of processes to one or two. Not all problems can be modelled in this fashion. For example, a lossy channel which never looses an infinite number of messages consecutively can be approximated by a process which reliably transmits a minimum of 1 message out of every k, for some fixed k. But this approach cannot be the basis for our model-checking induction since the k must range over an infinite set of values. Likewise, we cannot apply the technique directly to model our reservation protocol for an arbitrarily large resource set.

However end-to-end protocols are inherently inductive, in that they are designed to operate with arbitrary numbers of participating components. Provided that approximations for unbounded state variables are sufficient, the technique is very useful for proving properties of these protocols, such as livelock and deadlock freedom.

Again as in the first case-study described, in order for the formal technique to be effective (in this case, usable at all for the problem at hand), we had to first develop some techniques requiring specialist knowledge in the underlying theory. In this case, we had to make especially clever use of the CSP hiding operator for properly building an inductive structure, and we had to use "lazy abstraction" (previously used only for establishing security properties [Ros97]) to ensure that our checks were sufficiently strong.

5. Conclusions

I have described two case studies from my experience with computer networks which illustrate that all too often, existing techniques in our formal methods tool bags do not match industrial-strength problems. Encouragingly for formal methods advocates, with some extra work effective techniques were developed which solved the problem at hand. These techniques required specialist knowledge in the theory underpinning the formal method. Significantly, however, the theoretical basis for the formal semantics did not have to be extended or redefined in any way. Rather we had to appeal to the theory in order to establish soundness of the new techniques.

It is generally recognised that although there has been considerable work in formal methods involving theoretical foundations, standards, and even case-studies, industrial uptake of formal methods is low. Historically for computer networks such as the Internet, correctness potentially offered by formal methods has not been considered a problem; the Internet is characterised by best-effort rather than guaranteed service, and bug fixes have typically been cheap and easy (simply download an update from the Internet). However the success of such multiservice networks is bringing demands for such concerns as security and financial integrity, where establishing correctness is deemed essential.

If formal methods are to be effective for this new generation of network applications, it is essential that the methods are mature enough to be usable by people who are specialists in their application areas rather than in the formal theory. It is unrealistic to expect application specialists, even intelligent and knowledgeable ones, to have the expertise, time or inclination to

develop techniques requiring particular knowledge of the finer points of theory underpinning the formal method. The case studies described here illustrate that there remain significant technical challenges for practical use of formal methods which come to light only through application to realistic, large-scale problems. It continues to be important for formal methods specialists to apply the methods to a variety of industrial-strength problems, and make available any resultant techniques which contribute to the maturity and applicability of the methods.

References

[BC95] D Barnard and Simon Crosby, The Specification and Verification of an Experimental ATM Signalling Protocol, *Proc. IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, Dembrinski and Sredniawa, eds,Warsaw, Poland, June 1995, Chapman Hall.

[But92] R Butler. A CSP Approach to Action Sysems, DPhil Thesis, University of Oxford, 1992.

[BZB96] R Braden, L Zhang, S. Berson, S. Herzog and S. Jamin. Resource reSerVation Protocol (RSVP) – Version 1, Functional Specification. Internet Draft, Internet Engineering Task Force. 1996.

[Cre] S Creese, An inductive technique for modelling arbitrarily configured networks, MSc Thesis, University of Oxford, 1997.

[CR] S Creese and J Reed, Inductive Properties and Automatic Proof for Computer Networks, (to appear).

[Dav91] J Davies, Specification and Proof in Real-time Systems, D.Phil Thesis, Univ. of Oxford, 1991.

[FDR94] Formal Systems (Europe) Ltd. Failures Divergence Refinement. *User Manual and Tutorial*, version 1.4 1994.

[ftpe] Estelle Specifications, ftp://louie.udel.edu/pub/grope/estelle-specs

[GJ94] JD Guttman and DM Johnson, Three Applications of Formal Methods at MITRE, *Formal Methods Europe*, LNCS 873, M Naftolin, T Denfir, eds, Barcelona 1994.

[GPB91] R Groz, M Phalippou, M Brossard, Specification of the ISDN Linc Access Protocol for D-channel (LAPD), CCITT Recommendation Q.921, ftp://louie.udel.edu/pub/grope/estelle-specs/lapd.e

[Hoa85] CAR Hoare. *Communicating Sequential Processes.* Prentice-Hall 1985.

[ISOE] ISO Recommendation 9074, The Extended State Transition Language (Estelle), 1989.

[ISOL] ISO: Information Processing System - Open System Interconnection - LOTOS - A Formal Description Technique based on Temporal Ordering of Observational Behavior, IS8807, 1988.

[Jac96] DM Jackson. Experiences in Embedded Scheduling. *Formal Methods Europe*, Oxford, 1996.

[Jma95] M Jmail, An Algebraic-temporal Specification of a CSMA/CD Protocol, *Proc. IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, Dembrinski and Sredniawa, eds,Warsaw, Poland, June 1995, Chapman Hall.

[KR93] A Kay and JN Reed. A Rely and Guarantee Method for TCSP, A Specification and Design of a Telephone Exchange. *IEEE Trans. Soft. Eng.*. 19,6 June 1993, pp 625-629.

[LY93] G Leon, JC Yelmo, C Sanchez, FJ Carrasco and JJ Gil, An Industrial Experience on LOTOS-based Prototyping for Switching Systems Design, *Formal Methods Europe*, LNCS 670, JCP Woodcock and DG Larsen, eds., Odense Denmark, 1993.

[NM90] J Navarro and P s Martin, Experience in the Development of an ISDN Layer 3 Service in LOTOS, *Proc. Formal Description Techniques III*, J Quemada, JA Manas, E Vazquez, eds, North-Holland, 1990.

[PS91] K Paliwoda and JW Sanders. An Incremental Specification of the Sliding-window Protocol. *Distributed Computing.* May 1991, pp 83-94.

[R98] JN Reed, DM Jackson, B Deianov and GM Reed, Automated Formal Analysi of Networks: FDR Models of Arbitrary Topologies and Flow-Control Mechanisms, ETAPS-FASE98 European Joint Conference on Theory and Practice of Software; Fundamental Approaches to Software Engineering, Lisbon Portugal, March 1998.

[RGG95] AW Roscoe, PHB Gardiner, MH Goldsmith, JR Hulance, DM Jackson, JB Scattergood, H ierarchical compression for model-checking CSP or How to check 10^{20} dining philosphers for deadlock, Springer LNCS 1019.

[Ros97] AW Roscoe. *The CSP Handbook*, Prentice-Hall International, 1997.

[RR86] GM Reed and AW Roscoe, A timed model for communicating sequential processes, Proceedings of ICALP'86, Springer LNCS 226 (1986), 314-323; *Theoretical Computer Science* 58, 249-261.

[Shan] N Shankar, Machine-Assisted Verification Using Automated Theorm Proving and Model Checking, *Math. Prog. Meothodology*, ed. M Broy.

[Sid93] K Sidle, Pi Bus, *Formal Methods Europe*, Barcelona, 1993.

[Sin97] J Sinclair, Action Systems, Determinism, and the Development of Secure Systems, PHd Thesis, Open University, 1997.

[Tan96] AS Tanenbaum. *Computer Networks.* 3rd edition. Prentice-Hall 1996.

[TCSP92] J Davies, DM Jackson, GM Reed, JN Reed, AW Roscoe, and SA Schneider, Timed CSP: Theory and practice. *Proceedings of REX Workshop, Nijmegen*, LNCS 600, Springer-Verlag, 1992.

[Tur86] JS Turner. New Directions in Communications (or Which Way to the Information Age). *IEEE Commun. Magazine.* vol 24, pp 8 -15, Oct 1986.

[wwwl] LOTOS Bibliography, http://www.cs.stir.ac.uk/ kjt/research/well/bib.html

[ZDE93] L Zhang, S Deering, D Estrin, S Shenker and D. Zappala. RSVP: A New Resource ReSerVation Protocol. *IEEE Network*, September 1993.

A Framework for Evaluating System and Software Requirements Specification Approaches

Erik Kamsties and H. Dieter Rombach

Fraunhofer Institute for Experimental Software Engineering
Sauerwiesen 6, D-67661 Kaiserslautern, Germany
{kamsties, rombach}@iese.fhg.de

Abstract

Numerous requirements specification approaches have been proposed to improve the quality of requirements documents as well as the developed software and to increase user satisfaction with the final product. However, no or only anecdotal evidence exists about which approach is appropriate in a particular context. This paper discusses the value of experimentation in requirements engineering to gain sound empirical evidence. A framework is suggested which facilitates experimentation through an experimental infrastructure based on the QIP and GQM Paradigm. It helps to structure and formalize a research agenda for experimentation and supports the development of experiments. Our agenda and a set of experiments are outlined, focusing on requirements specification approaches for embedded real-time systems.

1 Introduction

Software developers who wish to improve either the productivity or the quality of the software they develop are faced with an enormous portfolio of techniques, methods, tools, and standards for requirements specification. However, no or only anecdotal evidence exist about which approach is appropriate in a particular context [Fen93]. Open questions are often: (1) under which conditions are requirements specification approaches profitable at all, (2) which approach should be applied in which type of project, (3) how can an approach be applied most efficiently, and (4) what is the impact on related activities (e.g., testing). A typical example for an anecdotal evidence is the following statement which was taken from the brochure of a CASE tool for requirements modeling: "the maintenance costs are lowered, the quality of your applications improves, and you deliver applications to market faster".

The transfer of software engineering technology to industry is plagued with a lot of problems, which can be attributed to some degree to the lack of empirical evidence. First, new technologies are often rejected by project personnel, since these are considered not well adapted to project needs and, thus, are perceived as not beneficial. Second, new technologies are bypassed under project pressure since project personnel are not convinced enough of the benefits to take any risks and they are not supported by the project management. Project progress is often measured only in lines of code by the project management. Third, past project experiences are not reused in new projects because benefits were not demonstrated explicitly and, thus, "religious" beliefs win [Rom97].

Empirical research provides strong methods, such as controlled experiments, to overcome the limitations of ad-hoc evaluation of software engineering technology. For instance, results from empirical research indicates that the introduction of a CASE tool actually leads to a decrease of

productivity in the first year [Gla92]. Controlled experiments have been proven to be a particularly effective means for evaluating software engineering methods and gaining the necessary understanding about their utility [LR96]. We will discuss their strengths and weaknesses in more detail in the next section. Moreover, an organization's software competencies are manageable assets. Software competencies are tailored technologies and methodologies that play a key role in supporting strategic capabilities of an organization[1]. Experiments are the key to building up software competencies [Rom97].

A brief introduction into methods for empirical research is given in the next section. The third section discusses the current practices used in evaluating requirements specification approaches, which are the focus of our agenda, and the benefits and drawbacks of experimentation in this area and in requirements engineering in general. The fourth section outlines the framework for experimentation and our agenda for the empirical investigation of requirements engineering. Examples of empirical studies found in the requirements engineering literature are characterized according to our agenda and an additional set of experiments is proposed.

The focus of our agenda is on *requirements specification approaches*, more precisely, on languages, techniques, and methods supporting the specification/documentation of requirements as well as on associated techniques to verify those requirements. It is worthwhile to investigate requirements specification approaches, since it is well-known that most of the defects found in software are caused by misconceptions in the requirements phase. Requirements specification approaches are proposed as one way of overcoming these problems. Only a few of them are applied in industrial practice. We believe that this is due to the relatively high investment compared to other improvements, for instance, the introduction of inspections, together with unproven merits. Religious beliefs in object-oriented approaches is one manifestation of this situation. Therefore, our long term goal is to provide empirical insights into this area of requirements engineering. A summary and an outlook on future work concludes this paper.

2 Introduction into Methods for Empirical Research

Software engineering and consequently requirements engineering is an amalgamation of influences from many fields including theoretical computer science, physical sciences, electrical engineering, behavioral and life sciences [Cur88]. Considering requirements engineering, for instance, formal methods [BBD+96] stem from research in theoretical computer science, while ethnography [SRS+93] has its roots in the behavioral and life sciences. In general, the parent research fields were often used as sources and inspirations for technology development in software engineering, but the underlying research methods of these fields were not adopted to a large extent. Software engineering and requirements engineering research is about developing languages, techniques, methods, and tools. Their validation did not play such an integral part of research as the confirmation and validation of models and hypotheses in physical, behavioral, or life sciences. It has been claimed that in software engineering, there is a lack of experimentation to validate research results [TLPH95]. Proposing a model or building a tool is not enough. There must be some way of validating that the model or tool is an advantage over current models or tools [Bas92]. There are some indications that this situation is beginning to change, for instance, the classification scheme used for submissions to the "International Symposium on Re-

1. Strategic capabilities are corporate goals defined by the business position of the organization and implemented by key business processes.

quirements Engineering" encourages researchers to perform either case studies (dimension D: 'Case study applying a proposed solution to a substantial example") or more objective evaluations, i.e., experiments (dim. E: 'Evaluation or comparison of proposed solutions') [Zav97].

We believe that software engineering is a true engineering task. Hence, improvement of practical software development requires an "experimental" approach. Basili outlines three research paradigms which comprise experimentation, namely the scientific method, the engineering method, and the empirical method [Bas92]. The engineering method and the empirical method are variations of the scientific method. All three methods are depicted in figure 1.

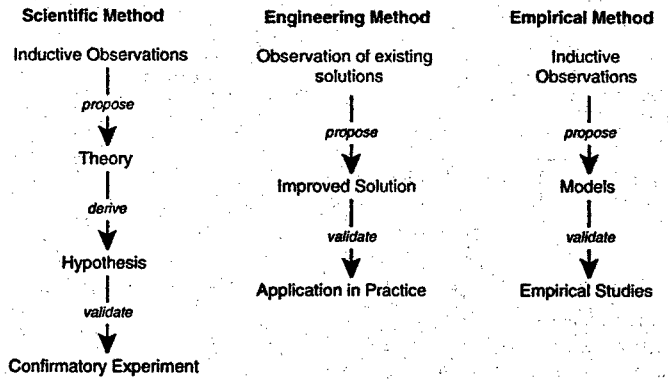


Figure 1: Research Methods

Basili recommends to apply the research paradigms as follows: "*In the area of software engineering the scientific method might best be used when trying to understand the software process, product, people, and environment. It attempts to extract from the world some form of model which tries to explain the underlying phenomena, and evaluate whether the model is truly representative of the phenomenon being observed. It is an approach to model building. (...) The engineering method is an evolutionary improvement oriented approach which assumes one already has models of the software process, product, people, and environment and modifies the model or aspects of the model in order to improve the thing being studied. (...) The empirical method is a revolutionary improvement oriented approach which begins by proposing a new model, not necessarily based upon an existing model, and attempts to study the effects of the process or product suggested by the new model.*" In an industrial context, the empirical method can help select and introduce a promising technique which is afterwards continually optimized by following the engineering method.

Experiments can be distinguished by several dimensions, namely, the purpose, the control over independent variables, the style of data, and the type of statistical analysis. The purpose of an experiment can be scientific learning, teaching, training, or technology/process evaluation and optimization. Examples include experiments to investigate the influence of domain knowledge on the efficiency of inspections, experiments at university to motivate software engineering principles, and industrial experiments to compare actual practices with new technologies in order to raise confidence. The degree of control over independent variables decides whether a controlled experiment or a case study is appropriate. A controlled experiment requires high lev-

el, and low difficulty of control, a case study must be performed otherwise. This decision has also an impact on the style of data gained and the type of statistical analysis. Controlled experiments are better in establishing causal relationships, while the conclusions from case studies are limited to the particular conditions of the study.

Experiments can be characterized further by the number of teams replicating each project and the number of different projects analyzed. A characterization scheme for the scope of investigation was developed by Basili et.al. [BSH86] which is depicted by table 1. Four different types of studies are outlined, namely blocked subject-project, replicated project, multi-project variation, and single project.

	1 Project	**> 1 Project**
1 Team	Single Project (Case Study)	Multi-Project Variation
> 1 Team	Replicated Project	Blocked Subject-Project

Table 1: Scope of Empirical Studies

Blocked subject-project studies examine one or more objects, i.e., the examined processes, products, or models, across a set of teams and a set of projects. Replicated project studies examine object(s) across a set of teams and a single project, while multi-project variation studies examine object(s) across a single team and a set of projects. Single project studies examine object(s) on a single team and a single project. Teams are possibly single-person groups that work separately, and projects are separate programs or problems on which teams work. As the scope of examination increases, the wider-reaching a study's conclusions become and the higher the cost. Small studies can be performed in a quantitative mode while larger studies typically involve more qualitative and less quantitative analysis.

Experimentation must be guided and there must be a rational for data collection, i.e., a framework for experimentation is required. Several frameworks have been proposed to design and analyze empirical studies in software engineering including DESMET by Kitchenham, Pfleeger, et.al. [MSL93], [Pfl95], [Kit97]. DESMET focuses on the evaluation of methods and tools, either in a qualitative (subjective), quantitative (objective), or hybrid mode through surveys, case studies, and formal (i.e., controlled) experiments. We suggest using the following components as an experimental infrastructure:

- **Quality Improvement Paradigm** (QIP) [BR88], [Bas89]
  The QIP provides an experimental framework for software development based on the scientific method (see figure 1). According to the QIP, projects within an organization are based on the continuous iteration of characterization, goal setting, selection of improved technology, monitoring and analysis of its effects to correct projects on-line, post-mortem analysis to understand what could be done better in future projects, and packaging the newly learned lessons so they can be reused efficiently in future projects.
- **Goal/Question/Metric Paradigm** (GQM) [BR88], [BCR94b], [BDR96]
  The GQM Paradigm supports the process of stating goals, refining goals in an operational way into metrics, and interpreting the resulting data. The idea behind the GQM Paradigm is that measurement (and hence experimentation) should be based on goals. By stating goals

explicitly, all data collection and interpretation activities are based on a clearly documented rationale.

- **Experience Factory** concept [BCR94a]
  The experience factory facilitates the reuse of models, gained for instance by experimentation, across project boundaries within an organization.

We will use this experimental infrastructure within our framework which is described in chapter 4.

3 Empirical Research in Requirements Engineering

Experimentation in requirements engineering (RE) was discussed in a panel session at the International Symposium on Requirements Engineering in 1995 [Rya95][1]. Experimentation was considered quite important, but nevertheless especially difficult to perform in RE research. One critique was that RE methods are not relevant objects for experimentation, since RE is in its essence about understanding and problem solving, and none of the present RE methods would support these tasks sufficiently. Thus, *"requirements engineering is about insight not experimentation"* [Jac95]. We subscribe to the first part of the statement. But as long as there are no real problem solving methods in RE, we have to apply the principles, techniques, methods, and tools that RE research has produced so far. Empirical research can contribute to RE by evaluating the truth of principles and the effectiveness of techniques, methods, and tools.

Another critique on experimental RE was that it is limited to small and unrealistic problems. This is true to some extent for replicated project and blocked subject-project treatments (see table 1). But multi-project variation treatments can be performed in realistic environments in a quantitative mode as the field study of El Emam et.al. [EQM96] illustrates. In this study, a model was developed which predicts the impact of user participation on the quality of RE service and on the quality of RE products in the presence of uncertainty. The model was tested using quantitative data of 39 real world software development projects from different organizations instead of using toy problems. The results indicates that as uncertainty increases, greater user participation alleviates the negative influence of uncertainty on the quality of RE service, and as uncertainty decreases, the beneficial effects on the quality of RE service of increasing user participation diminish. The interaction between user participation and uncertainty had no impact on the quality of RE products. Empirical research can contribute to RE by validating predictive models.

The most popular approaches of gathering evidence about requirements specification approaches, which are the focus of our investigations, are providing a sample specification of a common exemplar (e.g., library, ATM) or qualitative results of an industrial case study. However, the advantages and disadvantages of these two approaches must be judged from two perspectives: that of the RE researcher and that of the practitioner in an organization. The major value of a common exemplar is to advance research efforts [FFFvL97]. From a practitioner's perspective it is less valuable, instead, it is more likely a demonstration of existence (e.g., an ATM can be described with notation XYZ). Industrial case studies are valuable for both researchers and prac-

1. It is difficult to discuss experimentation in requirements engineering, since the term 'experimentation' is used differently in the literature. Some authors use it as a synonym for 'just trying out' other imply controlled experiments with it. We use the term here as comprising case studies as well as controlled experiments.

titioners, since they indicate whether an approach scales up and fits into the context. Their disadvantages are that they are expensive to perform, and that their results are context-dependent. Most case studies as well as exemplar studies are mainly qualitative. Controlled experiments complement exemplar and case studies because they produce quantitative results. They are useful for researchers since they can be replicated at different locations and varied in order to increase confidence in an requirements specification approach and to understand the influencing factors better. They are useful for practitioners since they can be used to gain confidence in new techniques before they are applied under project pressure. Experiments are increasingly performed in other areas of software engineering, e.g., inspections, or software maintenance.

We suggest *common experiments* in analogy to common exemplars for requirements engineering. Similar to a common exemplar, a common experiment is available to everyone for replication and variation. A common experiment is either an exemplar case study or a controlled experiment, dependent on the tackled RE activities, which is developed and conducted according to our experimental infrastructure outlined in section 1. Because upstream RE activities, e.g., elicitation, negotiation, and formalization of requirements, are creative, time-consuming, and less guidance is available, they require a case study approach for investigation, since the influencing factors are not under tight control. The downstream RE activities, e.g., reviews and testing, can be investigated by controlled experiments. The basis for both types of common experiments is a common exemplar which we supplement with (1) guidelines and procedures in order to make the usage of the exemplars more controlled, and (2) goal-oriented data collection procedures in order to make comparisons possible. Exemplar case studies differ from industrial case studies in that first, the effort is lower, and second, exemplar case studies are repeatable since common exemplars are used in a controlled way. Replications are important to increase evidence for requirements specification approaches and to compare approaches. Feather et.al. [FFFvL97] propose the use of 'requirements exemplars' (i.e., natural requirements) instead of 'specification exemplars' (i.e., tidied and simplified requirements) to study the upstream RE activities as well. We use both types of exemplars in our common experiments.

Furthermore, we propose *situated experiments* in addition to common experiments. A situated experiment is more convincing to practitioners because, in contrast to a common experiment, exemplars and/or processes are taken from their individual application domain. The situated experiment in combination with the common experiment allows the question of whether application domain knowledge plays a role in the efficient application of an approach to be factored out. This two step approach of empirical evaluation has been applied successfully in the area of inspections [LD97].

An industrial scale case study may be useful as a follow-up to a common experiment. The experimental results can "prove" the feasibility of requirements specification approaches in the small (replicated project, or blocked subject-project) and industrial scale case studies are performed afterwards to analyze whether the results scale up in a realistic environment (single project, or multi-project variation).

Our framework, which is outlined in the next section, comprises an experimental infrastructure based on the QIP and GQM Paradigm and suggests a set of common experiments for experimentation with requirements specification approaches. Situated experiments can be derived from tailoring common experiments to a particular industrial environment.

4 Framework

The purpose of our framework is to facilitate experimentation in software engineering; first, by means of structuring and formalizing a research agenda, and second, by an experimental infrastructure to design, execute, analyze, and package experiments driven by this agenda. The framework is comprised of the following three components:

- a roadmap (i.e., agenda) for experimentation,
- guidelines for the construction of experiments based on the QIP and GQM Paradigm, and
- a set of experiment descriptions. Each description contains a characterization of the environment in which the experiment took place, the goals, the hypothesis, a description of the investigated objects, the experimental design, the statistical analysis, the results, and the experiences gained.

The particular environment (e.g., university, company), in which the framework should be applied plays an important role. First, for defining a meaningful roadmap, and second, for interpreting the experimental results, since extrapolations are depending heavily on the representativeness of the sample. Therefore, several instantiations of the framework in different environments are desired in order to increase the range of conclusions. Figure 2 illustrates our framework.

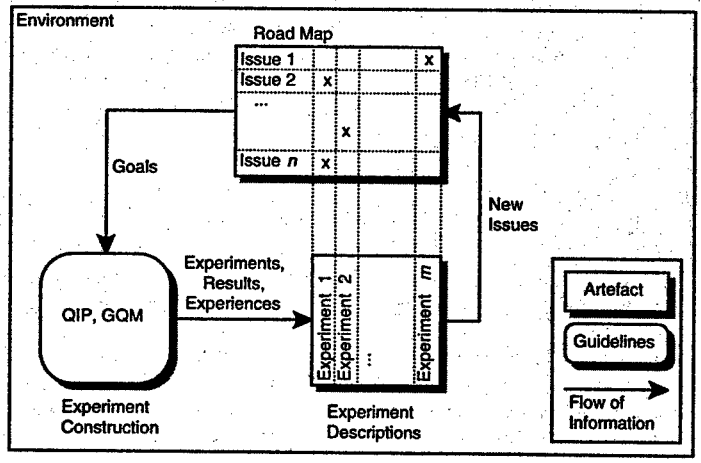


Figure 2: Framework

The components of our framework work together in the following manner: The roadmap raises a set of issues to be addressed by experimentation and maintains pointers to descriptions of already performed experiments. The 'experiment construction' component supports the design, execution, analysis, and packaging of new experiments based on the QIP and GQM Paradigm. The roadmap can be altered in case of new issues arising from the results of experiments conducted.

The roadmap and the experiment descriptions should be stored in an experience base (not illustrated in figure 2) in order to facilitate reuse [BCR94a]. A prototype implementation of an experience base which contains experiment descriptions, but not a roadmap, has been created within the Special Research Project 501 at the University of Kaiserslautern [FMV97], [FV98].

Compared to previously published frameworks for experimentation by Basili et.al. [BSH86] and Preece & Rombach [PR94] (see section 2), our framework provides additionally a "front-end", namely the roadmap, to describe a plan for a set of interrelated experiments. Guidelines for the construction of experiments and schemes to describe experiments are reused from the former frameworks.

The application of the framework in a particular environment involves three main steps: definition of a roadmap (i.e., research agenda), reuse of experiments and their results/experiences from comparable environments, if necessary, and construction of additional experiments. For instance, the Software Engineering Laboratory at NASA Goddard Space Flight Center performed a large number of experiments [BCM+92] which are candidates for reuse. The second objective of the framework, beside its application to construct experiments, is to classify experiments from the literature.

We applied our framework to requirements specification approaches for embedded systems in a university environment. Our agenda and a set of common experiments are described in the remainder of this section. We make both available to the RE community for:

- *Replicating experiments*
  A replication is an important contribution for two reasons. First, it helps validate the experimental design itself, and second, it increases potentially confidence in the previous results. For instance, several replications of the experimental comparison of testing versus code reading, e.g., [BS87], [KL95], [WRBM97] lead to sound evidence that code reading is as efficient as functional and structural testing.
- *Varying experiments*
  It is clear from the breadth of requirements engineering that no one researcher, or single research team, can be expected to solve all issues regarding the empirical evaluation of requirements specification approaches. Therefore, the experiments can be modified and re-executed, with the help of the experimental infrastructure, in order to increase and complement the empirical evidence gained so far.
- *Developing additional experiments*
  Goals for additional experiments can be derived directly from our agenda.

We discuss the framework components and our instantiation in the following subsections in more detail. The focus is on the roadmap and the suggested RE experiments.

4.1 Road Map

The roadmap represents a research agenda for a particular environment. It consists of (1) a set of *issues* (i.e., questions) which arise in a particular *environment* concerning a specific *theme*, (2) a "formalization" of issues in terms of *GQM goals* in order to characterize experiments precisely, and (3) pointer from issues to experiments. A GQM goal is defined by the following template:

> Analyze [object of study, e.g., products, processes, resources]
> for the purpose of [purpose of measurement, e.g., characterization, monitoring, etc.]
> with respect to [quality focus, e.g., cost, efficiency, etc.]
> from the viewpoint of the [viewpoint, e.g., researcher, practitioner, etc.]
> in the context of [description of specific context].

The facets describe *what* will be analyzed ("object of study"), *why* the object will be analyzed ("purpose"), *what property/attribute* of the object will be analyzed ("quality focus"), *who* uses the data collected ("viewpoint"), and in which environment the analysis take place ("context"). We propose the goal template to characterize empirical studies in requirements engineering.

The development of a relevant roadmap is a crucial task, since the issues must be relevant to the chosen viewpoint in order to create experiments which are of interest to a larger community within the particular environment. Our agenda (i.e., roadmap) for the investigation of requirements specification approaches is described in the remainder of this subsection.

Theme and Environment. We currently focus on issues regarding languages for requirements specification and techniques for reviewing those requirements specifications, rather than dealing with all imaginable issues in this area. As the results and experiences with RE experiments increase, we will expand the agenda. Our investigation takes place in a university context. We concentrate on embedded real-time systems and state transition-based approaches like Statecharts, because these approaches are the most widely applied in industry beside natural language, and because these are embodied in the increasingly popular object-oriented modeling approaches (e.g., OMT, ROOM, OCTOPUS). We assume that a textual requirements document ("customer requirements") is present and a detailed specification ("developer requirements") has to be written, either in natural language or with a requirements specification approach. Appropriate techniques for reviewing the detailed specification have to be chosen in both cases. Table 2 below depicts the issues of our current agenda and table 3 provides pointers to already existing experiments and to our proposals, respectively. The tabular representation does not imply a specific order of issues. Other representations, e.g. directed graphs, are possible.

Environment	Issue	Goal Definition				
		Object	Purpose	Quality Focus	Viewpoint	Context
See above	Informal vs. formal What are the differences of applying semi-formal or formal requirements specification approaches compared to relying entirely on textual requirements?	Requirements specifications	Evaluation	• Quality of requirements specification, e.g., understandability • Effectiveness and efficiency of requirements specification related activities, e.g. verification • Effort for further development processes and on quality of code • Effectiveness and efficiency of software testing (system/acceptance test, field test)	Researcher	University (e.g. Univ. of Kaiserslautern)
	Languages Which requirements specification language/approach is best suited for a particular environment?	Requirements specifications	Evaluation		Researcher	
	Review Which defect detection technique is most effective for requirements specifications written in a particular language?	Defect detection techniques	Evaluation		Researcher	

Table 2: Issues for Investigation of Requirements Specification Approaches

Issues. A set of issues (i.e., questions) arise in the environment described before:

- *Informal vs. formal.* What are the differences of applying semi-formal or formal requirements specification languages compared to relying entirely on textual requirements? The benefits of semi-formal and formal languages are often claimed: they help to avoid misunderstandings due to ambiguity, inconsistency, or incompleteness. Nevertheless a significant number of software development projects are conducted with completely informal requirements documents. There are several reasons for this situation, one of them is that there are doubts that the promised improvements can be reached in practice.
- *Languages.* Which language for requirements specification is best suited for a particular environment? A lot of requirements specification languages/approaches have been proposed in the past years. For instance, state-based approaches seem to be easy to apply since their theoretical foundation is often taught in lectures on computer science. Nevertheless, the expressiveness is somewhat limited (which is indicated for instance by the high number of extensions to statecharts). Therefore, more powerful languages have been proposed, for example, to describe timing properties in real-time systems more concisely.
- *Reviews.* Which defect detection technique is most effective for requirements specifications written in a particular language? Reviews of requirements specifications are important to ensure correctness and completeness with respect to the customer's needs. This is a complex task which cannot be automated. Several techniques have been proposed ranging from general purpose reading techniques to specialized techniques for requirements.

Each "informal" issue is "formalized" by a GQM goal as described in table 2. Furthermore, each issue/goal defines a set of possible experiments. We discuss in the following only the facets of the GQM goals, but not each goal in detail.

Object. The objects of study are either requirements specifications written in different languages, or defect detection techniques. There is a danger to compare "apples and oranges" since there are some requirements specification approaches such as Structured Analysis, in which the language is inherently coupled to a method. Others, such as Z are merely languages.

Purpose. The purposes of our empirical studies with requirements specification approaches are understanding, characterization, and evaluation.

Quality Focus. The quality focus is a quantitative description of subjective terms like "better" used in the description of an issue. A list of qualities is provided, covering the whole software lifecycle, which might be influenced by introducing a requirements specification approach. All these qualities are of interest for each issue, however some factors might be too expensive to measure.

Viewpoint. The viewpoint in our first investigations is that of the researcher. Other possible viewpoints include practitioner, manager, etc.

Context. The experiments have to be performed with students at the university. This limits the choice of objects because of the required training effort.

The issues raised before are used to characterize experiments found in the RE literature and to identify areas which up to now have been neglected or which warrant more investigation. Table 3 below lists some examples of experiments found in the RE literature as well as our proposal for an additional set of experiments. The studies are characterized according to the issue they address, the investigated objects, the quality focus, and the type of experiment (controlled ex-

periment / case study). The degree to which a study covers an issue and the open questions have to be summarized for each study (which is omitted from table 3).

Issue	Study	Objects	Quality Focus	Type
Informal vs. formal	Mills [Mil96]	Documents written in natural language and Real-Time Structured Analysis	Effectiveness of functional testing	Controlled experiment
Languages	**Our proposal**	**Documents written in SCR, OMT**	**Understandability, Testability, Verifiability, Modifiability**	**Controlled experiments**
Review	Porter et.al. [PVB95]	Adhoc, checklist, scenario-based reading	Effectiveness and efficiency of different reading techniques on SCR specifications	Controlled experiment

Table 3: Pointer to Experiments

4.2 Experiment Construction

General guidance for the design, execution, analysis of experiments with the GQM Paradigm is provided by the experimentation frameworks by Basili et.al. [BSH86] and by Preece & Rombach [PR94]. How to incorporate the QIP was discussed in [LM96]. It might be useful to create an instantiation of these frameworks with tailored and more detailed guidelines, if a topic requires extensive experimentation and experiences with experimentation in this area already exist. This was done for controlled experiments in the area of defect detection techniques by Lott et.al. [LR96].

Since there are only a few experiments in the area of requirements engineering, as indicated by table 3, and our own experiences with experimentation in RE are mainly concerned with inspections of textual requirements documents, we do not present detailed guidelines at this time.

4.3 Experiment Descriptions

Experiments are described according to Basili et.al. [BSH86] by four categories which reflect the phases of an empirical study:
1. Definition (i.e, study goals, scope, and hypotheses)
2. Planning (i.e, design, metrics, etc.)
3. Operation (i.e, data collection, validation, analysis)
4. Interpretation

Similar to the discussion of experiment construction above, it might be useful to refine these categories to particular empirical approaches or particular topics of investigation as it was done in [LR96] for controlled experiments in the area of defect detection techniques.

We do not present a more detailed characterization scheme for the same reasons as discussed in section 4.2 "Experiment Construction". In the remainder of this subsection our proposal for experiments in RE is presented according to the above scheme.

We have started to define and perform a first set of controlled experiments regarding the evaluation of different languages provided by requirements specification approaches. This issue has been addressed so far merely by qualitative case studies. We used initially the Unified Modeling

Language (UML) [Rat97] together with the OMT method [RBP+91] and the SCR style tabular requirements technique (SCR = Software Cost Reduction) [vS92]. The motivation for these experiments is two-fold. First, object-oriented requirements analysis (OORA) approaches are becoming more and more popular. They claim to facilitate understanding, since objects map directly to real-world entities. OMT is the most applied OORA approach, which is especially used in technical domains. However, since the behavioral specification in OMT is distributed over a set of collaborating objects, it is not easy to tell whether an analysis model satisfies the required end-to-end behavior. The UML is the de-facto standard for documenting object-oriented models. Second, SCR is a widely-known black-box specification technique for technical domains (embedded real-time systems) where its tabular notation is claimed to be readily understandable. It is easy to verify, since system-internals are not described. Nevertheless, complexity inherent to large systems cannot be wished away by methodological choices and SCR specifications become complex, too.

Definition.
Goal: Analyze **techniques T_1, T_2, ..., T_n to express requirements**
- T_1: UML language with OMT method
- T_2: SCR style tabular requirements technique
- T_n: further requirements specification approaches

for the purpose of **evaluation**
with respect to the **understandability, verifiability, testability, and modifiability**
from the viewpoint of the **researcher**
in the context of **a lab course/lecture at the University of Kaiserslautern.**

Scope: Replicated project or blocked subject-project
Hypothesis: There is a difference with respect to (a) *understandability,* (b) *verifiability,* (c) *testability,* (d) *maintainability.*

Plan.
Objects: Documents in T_1 and T_2
Subjects: Students / Practitioners
Tasks:
- Answer questions regarding behavioral and functional aspects of the specification (a)
- Check completeness and consistency with respect to informal requirements (b)
- Perform changes on the specification (c)
- Design test cases based on the specification (d)

Independent and dependent variables:

Independent Variables	Dependent Variables	
• Run (run 1 and run 2) • Requirements Specification Approach (T_1, T_2) • Type of Document • Experience of Subjects	Understandability measured via questionnaire	• Time needed to complete the questionnaire • Correctness of answered questions • Completeness of answered questions
	Verifiability measured via defect form	• Number of inconsistencies found • Time needed
	Testability measured via test cases	• Time required to write test cases

Table 4: Overview on independent and dependent Variables

Independent Variables	Dependent Variables	
see above	Modifiability measured via success of changes	• Time needed to perform changes • Completeness of changes • Correctness of changes • Modification rate

Table 4: Overview on independent and dependent Variables

Design (random assignment to groups A, B):

Run	Technique (Document)	
	T_1 (D_1)	T_2 (D_2)
1	A	B
2	B	A

Table 5: 2 x 2 Within-subjects factorial design

The 'operation' and 'interpretation' categories are omitted in this experiment description since this is only a proposal. These experiments are open for variation. It should be relatively easy to exchange the used specifications, or both the employed requirements specification approaches and the used specifications.

5 Summary and Future Work

In this paper we have discussed the role of experimentation in requirements engineering in overcoming the lack of empirical evidence in the field. A framework for experimentation in software engineering, not only requirements engineering, was suggested which facilitates experimentation by means of structuring and formalizing a research agenda (i.e., roadmap), and by an experimental infrastructure for developing experiments according to this agenda. We have presented our agenda for experimentation with requirements specification approaches for embedded real-time systems, and proposed a set of experiments. The agenda was also used to characterize already existing experiments in the literature. We performed one of the proposed experiments at the University of Kaiserslautern in December 1997 [KvKR98]. Currently, we are capturing experiences made in other environments–e.g., the NASA Software Engineering Laboratory–, developing further experiments, and extending our agenda. The proposed framework, the agenda, and the experiments are parts of a PhD thesis that is currently being performed at the Fraunhofer Institute (IESE).

Performing experiments in requirements engineering is beneficial for students, practitioners, and the research community. Students can experience the relative strengths and weaknesses of the requirements engineering approaches that are introduced in their lectures. Professionals can gain confidence in new approaches before they are applied under project pressure. The research community can accumulate a body of knowledge regarding the utility of various approaches under varying project characteristics. We therefore recommend that replicable experiments be adopted as a standard part of both education and technology transfer programs.

The empirical evaluation of requirements engineering approaches cannot be the effort of a single person or a single research group. Many experiments are too large for any single organization, they must be repeated in different environments. The International Software Engineering Research Network (ISERN) is a community that believes software engineering research needs to be performed in an experimental context. ISERN has facilitated the development of experiments and their replication in different environments. Organizations interested in joining ISERN may access the World-Wide Web information available from the following URL

`http://wwwagse.informatik.uni-kl.de/ISERN/isern.html`

or send an email to

`isern@informatik.uni-kl.de`.

6 Acknowledgments

We thank our colleagues in the Software Engineering Research Group at the University of Kaiserslautern and in the Fraunhofer Institute for their contributions, especially Colin Atkinson, Jean-Marc DeBaud, and Khaled El Emam for their comments on previous versions of this paper.

7 Literature

[Bas89] Victor R. Basili. Software development: A paradigm for the future. In *Proceedings of the 13th Annual International Computer Software and Application Conference (COMPSAC)*, pages 471–485, Orlando, Florida, September 1989.

[Bas92] Victor R. Basili. The experimental paradigm in software engineering. In H. D. Rombach, V. R. Basili, and R. W. Selby, editors, *Experimental Software Engineering Issues: A critical assessment and future directions*, pages 3–12. Lecture Notes in Computer Science Nr. 706, Springer–Verlag, September 1992.

[BBD+96] Jonathan P. Bowen, Ricky W. Butler, David L. Dill, Robert L. Glass, David Gries, Anthony Hall, Michael G. Hinchey, C. Michael Holloway, Daniel Jackson, Cliff B. Jones, Michael J. Lutz, David Lorge Parnas, John Rushby, Hossein Saiedian, Jeannette Wing, and Pamela Zave. An invitation to formal methods. *IEEE Computer*, 29(4):16–30, April 1996.

[BCM+92] Victor Basili, Gianluigi Caldiera, Frank McGarry, Rose Pajersky, Gerald Page, and Sharon Waligora. The Software Engineering Laboratory – an operational Software Experience Factory. In *Proceedings of the 14th International Conference on Software Engineering*, pages 370–381, May 1992.

[BCR94a] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. Experience Factory. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 469–476. John Wiley & Sons, 1994.

[BCR94b] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. Goal Question Metric Paradigm. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 528–532. John Wiley & Sons, 1994.

[BDR96] L. Briand, C. Differding, and D. Rombach. Practical guidelines for measurement-based process improvement. In *Proceedings of the International Software Consulting Network Conference (ISCN'96)*, 1996.

[BR88] Victor R. Basili and H. Dieter Rombach. The TAME Project: Towards improvement–oriented software environments. *IEEE Transactions on Software Engineering*, SE-14(6):758–773, June 1988.

[BS87] Victor R. Basili and Richard W. Selby. Comparing the effectiveness of software testing techniques. *IEEE Transactions on Software Engineering*, 13(12):1278–1296, December 1987.

[BSH86] Victor R. Basili, Richard W. Selby, and David H. Hutchens. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, SE-12(7):733–743, July 1986.

[Cur88] Bill Curtis. A methodological and empirical basis for research in software engineering. Technical report, Microelectronics and Computer Technology Corporation, 1988.

[EQM96] Khaled El Elmam, Soizic Quintin, and Nazim H. Madhavji. User participation in the requirements engineering process: An empirical study. *Requirements Engineering Journal*, 1(1):4–26, 1996.

[Fen93] Norman Fenton. How effective are software engineering methods? *Journal of Systems and Software*, 22(2):141–146, August 1993.

[FFFvL97] Martin S. Feather, Steven Fickas, Anthony Finkelstein, and Axel van Lamsweerde. Requirements and specification exemplars. *Automated Software Engineering*, 4(4):419–438, October 1997.

[FMV97] Raimund L. Feldmann, Jürgen Münch, and Stefan Vorwieger. Experiences with systematic reuse: Applying the EF/QIP approach. In *Proceedings of the European Reuse Workshop*, Brussels, Belgium, November 1997.

[FV98] Raimund L. Feldmann and Stefan Vorwieger. The web-based Interface to the SFB 501 Experience Base. SFB-501-TR- 01/1998, Sonderforschungsbereich 501, Dept. of Computer Science, University of Kaiserslautern, 67653 Kaiserslautern, Germany, 1998.

[Gla92] Robert L. Glass. Do measuring advocates measure up? In *Proceedings of the 3rd International Conference on Applications of Software Measurement*, pages 1.02–1.12, 1992.

[Jac95] Michael Jackson, 1995. Concluding statement at the panel session on "Let's have more Experimentation in Requirements Engineering" at the International Symposium on Requirements Engineering (RE'95).

[Kit97] Barbara Ann Kitchenham. Evaluating software engineering methods and tools. Parts 1 to 8. *ACM SIGSoft Software Engineering Notes*, 1996 and 1997.

[KL95] Erik Kamsties and Christopher M. Lott. An empirical evaluation of three defect-detection techniques. In W. Schäfer and P. Botella, editors, *Proceedings of the 5th European Software Engineering Conference*, pages 362–383. Lecture Notes in Computer Science Nr. 989, Springer–Verlag, September 1995.

[KvKR98] Erik Kamsties, Antje von Knethen, and Ralf Reussner. An empirical evaluation of two requirements specification techniques. Technical report, University of Kaiserslautern, Kaiserslautern, Germany, 1998.

[LD97] Oliver Laitenberger and Jean-Marc DeBaud. Perspective-based reading of code documents at Robert Bosch GmbH. *Information and Software Technology*, 39:781–791, 1997.

[LM96] Oliver Laitenberger and Jürgen Münch. Ein Prozeßmodell zur experimentellen Erprobung von Software-Entwicklungsprozessen. SFB-501-TR- 04/1996, University of Kaiserslautern, Special Research Project 501, 1996. In german.

[LR96] Christopher M. Lott and H. Dieter Rombach. Repeatable software engineering experiments for comparing defect-detection techniques. *Journal of Empirical Soft-*

ware Engineering, 1996.

[Mil96] Kevin L. Mills. An experimental evaluation of specification techniques for improving functional testing. *Journal of Systems and Software*, 32(1):83–95, January 1996.

[MSL93] W-E.A. Mohamed, C. J. Sadler, and D. Law. Experimentation in software engineering: A new framework. In *Proceedings of Software Quality Management '93*, pages 417–430. Elsevier Science, Essex U.K., 1993.

[Pfl95] Shari Lawrence Pfleeger. Experimental design and analysis in software engineering. Parts 1 to 5. *ACM SIGSoft Software Engineering Notes*, 1994 and 1995.

[PR94] Jenny Preece and H. Dieter Rombach. A taxonomy for combining software engineering and human-computer interaction measurement approaches: Towards a common framework. *International Journal of Human–Computer Studies*, 41:553–583, 1994.

[PVB95] Adam A. Porter, Lawrence G. Votta, and Victor R. Basili. Comparing detection methods for software requirements inspections: A replicated experiment. *IEEE Transactions on Software Engineering*, 21(6):563–575, June 1995.

[Rat97] Rational Software Corporation. *Unified Modeling Language*, 1997. Version 1.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Rom97] H. Dieter Rombach. Experimentation as a vehicle for software technology transfer: A family of software reading techniques. In *Proceedings of the 1st International Conference on Empirical Assessment & Evaluation in Software Engineering*, Keele (UK), March 1997. Keynote talk.

[Rya95] Kevin Ryan. Let's have more experimentation in requirements engineering. In *Proceedings of the 2nd IEEE International Symposium on Requirements Engineering*, page 66, York, U.K., March 1995. Panel session.

[SRS+93] Ian Sommerville, Tom Rodden, Pete Sawyer, Richard Bentley, and Michael Twidale. Integrating ethnography into the requirements engineering process. In *Proceedings of the IEEE International Symposium on Requirements Engineering (RE93)*, pages 165–173, San Diego, California, USA, January 1993.

[TLPH95] Walter F. Tichy, Paul Lukowicz, Lutz Prechelt, and Ernst A. Heinz. Experimental evaluation in computer science: A quantitative study. *Journal of Systems and Software*, 28(1):9–18, January 1995.

[vS92] A. John van Schouwen. The A-7 requirements model: Re-examination for real-time systems and an application to monitoring systems. CRL Report No. 242, McMaster University, CRL, Telecommunications Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, February 1992.

[WRBM97] Murray Wood, Marc Roper, Andrew Brooks, and James Miller. Comparing and combining software defect detection techniques: A replicated empirical study. In Mehdi Jazayeri and Helmut Schauer, editors, *Proceedings of the 6th European Software Engineering Conference – ESEC/FSE '97*, Lecture Notes in Computer Science No. 1301, pages 262–277, Zurich, September 1997. Springer.

[Zav97] Pamela Zave. Classification of research efforts in requirements engineering. *ACM Computing Surveys*, 29(4):315–321, 1997.

Postmodern Software Design with NYAM: Not Yet Another Method

Roel Wieringa
Department of Computer Science
University of Twente
the Netherlands
roelw@cs.utwente.nl*

Abstract

This paper presents a toolbox for software specification and design that contains techniques from structured and object-oriented specification and design methods. The toolbox is called TRADE (Toolkit for Requirements and Design Engineering). The conceptual framework of TRADE distinguishes external system interactions from internal components. External interactions in turns are divided into external functions, behavior and communication. The paper shows that structured and OO analysis offer a small number of specification techniques for these aspects, most of which can be combined in a coherent software design specification. It is also shown that the essential difference between structured and object-oriented software design approaches lies in the separation of data storage, data processing and control in data flow diagrams, versus the encapsulation of these into objects by OO analysis. Functional and subject-domain-oriented decomposition, on the other hand, are shown to be compatible with both approaches.

1 Introduction

In this paper, we view design as uncertainty reduction about the future of an artifact. In this broad view, design decisions may concern external properties as well as the internal structure of new artifacts, and may concern changes to existing artifacts or may concern new artifacts. The result of design is a documentation of the decisions made about the artifact, called a **specification**. Specifications always consist of a combination of text and graphics, both of which may vary in degree of formality. Software design specification methods are grouped into two camps. Structured methods emphasize data flows and functional decomposition. Object-oriented methods emphasize the encapsulation of operations and data and recommend something that we refer to as subject-domain-oriented decomposition.

*Work done while at the Faculty of Mathematics and Computer Science, *Vrije Universiteit*, Amsterdam.

The need for integration of structured and object-oriented software specification methods has been long recognized [1, 3, 32, 47, 51]. One of the advantages of such an integration is that it allows practitioners raised in the world of data flow modeling and functional decomposition to incorporate useful elements of object-oriented specification in a stepwise, evolutionary manner. Another advantage is that it allows us to pick the best elements of both groups of methods in an eclectic way, which should allow us to advance the state of the art beyond each of the contributing groups of methods. For example, we show that structured analysis offers useful techniques for the specification of external functionality but is a bit muddled in its specification of internal decomposition. Object-oriented analysis, on the other hand, offers useful techniques for the specification of decompositions but tends to ignore the specification of external functionality.

Early integration proposals incorporate object-oriented ideas in structured analysis without fundamentally changing structured analysis [51], or incorporate structured analysis in object-oriented analysis [3], or simply use structured analysis as a front-end to object-oriented design [1, 32, 47]. None of these proposals is based upon a thorough analysis of the underlying principles of structured and OO analysis. Without such an analysis, it is not possible to see which elements of structured and OO methods can or cannot be combined and why this is so. The integration proposed in this paper is based upon a thorough survey and analysis of six structured and 19 object-oriented specification methods, recently completed [57].

The results of the analysis are used to define the Toolkit for Requirements and Design Engineering (TRADE). This is a kit of *conceptual* tools, not software tools. TRADE contains techniques and heuristics taken from many different methods and allows combination of these tools in a coherent way. There is a software tool called TCM that can be used to use some of the techniques in TRADE, but TCM is not described in this paper, because the essential design tools are made from software but from the experience and understanding of the designer. The essential tools are conceptual and consist of design techniques and heuristics and the tacit knowledge needed to apply them. This approach is postmodern in the sense that TRADE contains only elements borrowed from existing methods. It adds nothing except a framework in which these elements are put, and set of rules for using these elements in a coherent way. I hope that TRADE will not be viewed as yet another method but as a toolkit that, as any other toolkit, should be used flexibly and in a context-sensitive way.

I start in section 2 with setting out a framework for software design methods that allows us to analyze structured and object-oriented methods and their techniques in a coherent manner. This framework is explained and motivated at length elsewhere [56]. In addition to allowing us to understand and analyze the use that can be made of specification techniques, the framework also allows us to define the relationships that must hold between the different techniques in a coherent multi-perspective specification. In section 3, a catalog is given of techniques taken from structured and object-oriented methods, and show how they fit into this framework. In section 4, it is shown which techniques are adopted in TRADE, and how these are connected. Section 5 concludes the paper with a discussion.

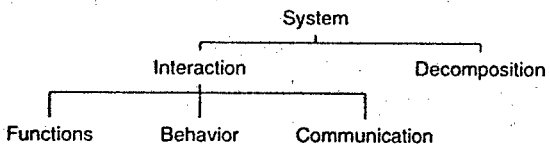


Figure 1: A framework for systems.

2 A Framework for Software Systems

The TRADE framework classifies the kinds of properties of software systems that a designer might want to specify. It ignores the design process but focusses on the system: we return briefly to this below. The framework is derived from frameworks for systems engineering [17, 18] and product development [45] and from an analysis of software design methods [57]. The two basic dimensions of the framework are those of external interactions and internal (de)composition (figure 1).[1] Each system interacts with its external environment and is viewed as part of an aggregation hierarchy, in which higher-level systems are composed of lower-level systems. External interactions and internal decomposition are orthogonal in the sense that design decisions about these two dimensions of a system can be separated.

The external interactions of a system should be useful for at least some other systems in its external environment (people, hardware or software). This means that we should always be able to partition external interactions into chunks of useful interactions that we call **external functions**. These chunks may be atomic from an external point of view (i.e. they are external *transactions*), or they may be complicated dialogs between the system and some external entities. They are not to be confused with mathematical functions or with functions written in a programming language. They are similar to Jacobson's [31, 40] use cases: pieces of external behavior that have some use for an external agent. Of the many properties that external functions can have, we single out two kinds: the ordering of functions in time, called **behavior**, and their ordering in "space", called **communication**. An external function is an external interaction, and each external interaction involves communication with one or more external entities. Moreover, external interactions are usually governed by rules of temporal precedence, which leads to the concept of behavior. The distinction between behavior and communication is the same as the classification of process operators in CCS into dynamic and static ones [38].

There should be a safety valve in our framework in the form of a category "all other properties". This includes the famous "ilities" such as usability, portability, interoperability, reliability etc. Many of these can be construed as properties of interactions or of the decomposition. However, structured and object-oriented methods provide no techniques or heuristics for specifying these properties and TRADE contains no tools for this, so we ignore this category in this paper.

[1] The terms "composition" and "decomposition" used in the paper do not refer to bottom-up or top-down design processes but are used to refer to the internal structure of a system.

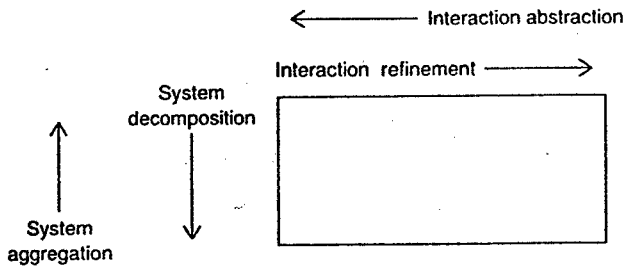


Figure 2: The magic square.

This completes the sketch of our framework for techniques. We will classify the techniques used in structured and object-oriented methods as techniques for specifying external functions, behavior or communication, or internal decomposition of a system. Before we do that, we point out a number of special features of the framework and show how it can be applied to software systems.

First, observe that each component of a system is itself a system. that has an internal decomposition and interacts with other components and external entities of the system. In fact, each interaction of the entire system is realized by interactions of its components. In terms of specification techniques, this means that we can use the same technique to specify interaction of systems at different levels in the aggregation hierarchy.

Next, we can specify a system's external external interactions at several levels of refinement, where higher levels of refinement correspond to more detail and less abstraction. We can also specify a system's components, components of those components, etc., leading to an aggregation hierarchy. The orthogonality of external interaction and decomposition implies that interaction refinement and system decomposition are orthogonal. This is visualized in figure 2, called the **magic square** by Harel and Pnueli [24]. Orthogonality means that decisions about interactions can be intertwined with decisions about decompositions in any way [49].

To explain this further, we return briefly to the process dimension. It is useful to distinguish logical design tasks from the way these tasks are ordered in time. Very generally, for any design task, the logical tasks are

- analysis of problem situation,
- synthesis of proposed solutions,
- simulation of solutions, and
- evaluation of simulations [45, 56].

For example, we may refine an interaction specification by analyzing the needs of the external environment, proposing a refinement, simulating the specified interactions, and evaluating this simulation. Or again, we may design a decomposition by analyzing the desired interactions of the system, proposing a decomposition, simulating its behavior, and

evaluating this simulation. During an actual project, these tasks may be ordered in time in various ways. For example, in waterfall development, the entire set of external interactions is specified before the system is decomposed in a top-down way. This is characterized by a path through the magic square that starts at the upper left corner and then proceeds in a top-down way by alternately moving right (refining) and down (decomposing). In incremental development, only the most important external interactions are specified before a system architecture is determined. The corresponding path through the magic square starts at the upper left corner and moves sufficiently right and down to determine the overall functionality and architecture of the system. It then performs a linear process for each increment of the system. Other strategies are possible too [24, 56, 60]. In each strategy, decisions about interactions and architectures are intertwined in a particular way. In whatever way this is done, the result of these decisions must be justifiable as if they were taken by means of a rational design process [41]. This is the design analogy to the way in which the historical progress of scientific knowledge can be rationally reconstructed as if a rational, empirical discovery procedure were followed [33, 56].

So far, the framework does not refer to special properties of software systems and is therefore applicable to all kinds of design. In the case of software systems, we add two features to our framework that will turn out to be useful to understand the use that is made of specification techniques. First, each software system interacts with its environment by exchanging symbol occurrences with its external entities. Now, a symbol occurrence is a physical item to which people have assigned a meaning. So for these people, it refers to part of the external world. I call the part of the world referred to by the external interactions of a software system the **subject domain** of the system. (Another term often used is *Universe of Discourse*.) The subject domain is itself a system and may itself be under development by another design team. So the framework of figure 1 is applicable to it. To understand how techniques are used in methods, it is important to understand what they are used for: to specify the subject domain or to specify the software system.

The second feature to be added to our framework is the identification of the essential level of aggregation in the specification of software systems. Given a specification of external functions, behavior and communications of a software system, we can design a decomposition of this system that would be optimal for this specification of external properties, and that ignores the properties of underlying implementation layers. I call this an **essential decomposition** of the software system. The only decomposition criteria that can be used for the essential decomposition are derived from the external environment of the system, such as its external functionality, external behavior, external communications, or its subject domain. The concept of essential decomposition arose with McMenamin and Palmer [37] and also occurs in the object-oriented method Syntropy under the guise of the specification model [10]. I return to this when we discuss structured and object-oriented decomposition criteria in section 3.2.2. All other decomposition levels of a software system are designed by taking aspects of the underlying implementation environment into account. For example, in a distributed system, the essential decomposition must be allocated to processors in the network, and at each processor, essential components must be mapped to schedulable sequential processes. I call these decomposition levels **implementation-oriented**.

3 A Catalog of Techniques

3.1 External interaction specification

3.1.1 External functions

In our framework, external interactions are partitioned into useful portions called functions. Functions can be organized in a refinement hierarchy such that the root of the hierarchy is the overall system function, called the **mission** of the system, and the leaves are **elementary functions**. I regard a function elementary if it is triggered by an event and includes all desired responses to the event. Borrowing from structured analysis, we distinguish **external events**, which arise from an external entity, from **temporal events**, which consist of a significant moment in time [52]. For example, pushing an elevator button is an external event. If the elevator doors have been open for a certain amount of time, a timeout occurs, which is a temporal event.

Techniques for specifying external functions come mainly from structured analysis.

- Often forgotten but extremely important is the **mission statement** of the system. In the Yourdon Systems Method (YSM) this is called the statement of purpose [66]. It consists of a general description of one or two sentences, the major responsibilities of the system and a list of things the system is agreed *not* to do.
- The external function hierarchy can be represented by a **function refinement tree** whose root represents the mission and the leaves represent the elementary functions. This is a well-known technique from Information Engineering [34]. The tree is merely an organization of external functions and does not say anything about the internal decomposition of the system.
- Elementary functions, which are at the leaves of the tree, can be represented as a list of **event-response pairs**, another technique from YSM, in which the source of the event, its meaning, the desired response of the system, timing requirements and other relevant externally observable properties are described [65, 66].
- If the data interface of events and responses is important, their **pre– and post-conditions** in terms of input and output data can be specified. This technique is used in structured and object-oriented analysis alike [9, 65, 66].

There are several techniques to specify behavioral and communication properties in structured and object-oriented analysis, discussed next. Many of these techniques are too detailed to be used for a specification of external system behavior at the higher levels of aggregation but as we specify the requirements of lower-level components, they become increasingly useful. These lower-level components are systems in their own right and we continue to refer to them as such.

3.1.2 Behavior

Two groups of behavior representation techniques are used in structured and object-oriented analysis, state transition diagrams (STDs) and process dependency diagrams.

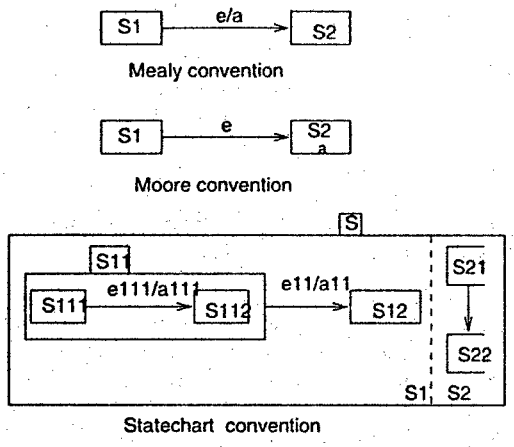


Figure 3: The Mealy, Moore and statechart conventions.

- State transition diagrams come in several flavors. In all flavors, an STD is a directed graph in which the nodes represent states and the edges state transitions. In the **Mealy convention**, an edge can be labeled by the *event* that triggers the transition and the *action* generated by the transition. The triggering event is an external event received from the environment of the system being specified and the generated action is the response sent to the environment. For example, in the Mealy STD of figure 3, if the system receives event e when it resides in state S1, it will generate a and move to state S2. Most structured analysis methods of the Yourdon school use Mealy STDs.

- In the **Moore convention**, actions must be associated with states rather than transitions. In the Moore STD of figure 3, if the system receives event e when it resides in state S1, it will move to state S2 and upon arrival in this state, generate a. The Shlaer-Mellor method for object-oriented analysis uses Moore STDs [48]. Formally, the Mealy and Moore representations have the same expressive power [26, page 42], because they recognize the same language.

- In the **Statechart convention**, both the Mealy and Moore conventions are allowed [19, 27]. More importantly, statecharts allow the representation of state hierarchies and parallelism. For example, in the statechart of figure 3, the system is represented by state S, which is partitioned into two parallel substates S1 and S2, each of which are further divided into substates. When the system is in state S111, it is also in state S11, in state S1 and in state S. If it receives event e11 when it is in state S111, it will leave S11 and S11, generate a11 and move to state S12. The generated action a11 may be sent to the external environment but may also be broadcast to all parallel components (such as S2). The execution semantics of statecharts is complex and has been studied in detail [4, 25]. Depending upon the execution semantics, a statechart can be replaced by a set of Mealy (or Moore) STDs

that communicate via broadcasting. Most object-oriented methods use statecharts to represent behavior. This may give the impression that the use of statecharts is the hallmark of object-oriented specification. This is a false impression, because Statemate, which is a structured approach, also uses statecharts [22, 27]. The Statemate execution semantics of statecharts is precisely defined [23], but the statechart versions used in object-oriented methods do not have a formally defined semantics. The exception is the UML, for which a formally defined execution semantics is currently being defined [44, 20, 21]. The reader should be warned that this semantics is totally different from the Statemate semantics.

- In order to be able to draw an STD, the number of represented states in an STD must be finite and small. The number of representable states can be increased if we introduce **local variables**. This requires an extension of the graphical technique with the ability to define data types and declare variables. Furthermore, edge labels must be extended with the possibility to specify **guards** that test the value of these variables, and with the possibility to specify additional actions that consist of assignments to these variables. If a guard evaluates to false when an event occurs, then the event e will not trigger the transition. An STD with local variables is called an **extended STD**. Mealy STDs, Moore STDs and statecharts can all be extended this way. Yourdon-style structured analysis do not use extended STDs, but other structured methods such as JSD [29] and SDL [5] and all object-oriented methods use extended STDs.[2] I argue below that this lies at the heart of the difference between these approaches.

- A small number of methods use **process dependency diagrams** to represent behavior. These are directed graphs in which the nodes represent processes and the edges process dependencies. The process at the tail of an arrow must have terminated in order for the process at the head to begin. Process dependency diagrams are typically used to represent the flow of control through a number of processes, each of which may be executed by a different system. They are used for example in workflow modeling. Process dependency diagrams were introduced in Information Engineering [35]. Martin and Odell use an expanded form of the notation in their object-oriented specification method [36] and yet another form of the notation is adopted in the UML [43]. There is as yet no formal semantics for these notations in their complete forms.

Of the STD techniques, Mealy and Moore representations are alternatives and statecharts are a more powerful variation. Extended versions of these techniques are more expressive than nonextended ones. Clearly, the STD techniques are not mutually incompatible but in a particular modeling effort, one of them should be chosen. Whatever STD convention is used, it assumes that external functions of the system have been specified as event-response pairs. The STD can then be used to represent temporal orderings of the event-response pairs.

[2] I regard JSD as a hybrid method that combines elements of structured and OO analysis. JSD uses process structured diagrams with accompanying text, that are equivalent to STDs. SDL is a description technique used for telecommunication systems based upon extended finite state machines. The design philosophy of SDL is functional decomposition but recently, object-oriented features have been added [7, 39].

Process dependency graphs do not require that functions have been specified as event-response pairs. Viewing each function as a process, the process dependency graph can then be used to represent temporal precedence relationships between these. Most varieties of process dependency graphs have no formal semantics. France [16] defines a formal semantics for data flow diagrams extended with control constructs, but the best of my knowledge, this has not been used in the reviewed methods. Because in the UML, process dependency diagrams (called activity diagrams) are based upon statecharts, there is hope that a formal statechart semantics can be used to define a formal semantics for activity diagrams. Until such a formal semantics is defined to decide the matter, an integrated approach should use STDs and not process dependency diagrams.

3.1.3 Communication

The following communication specification techniques are used in structured and object-oriented analysis at the system level.

- Very useful to represent the system boundary is the **context diagram**, which represents the system and the external entities which it communicates with (and which are sources of external events or destinations of responses). This is an important technique from structured analysis [14], recently reinstated by Jackson [30].
- A variant of the context diagram, introduced in Objectory and since adopted by many object-oriented methods, is the **use case diagram** [8, 31, 42]. A use case is an interaction between the system and its environment that is useful for its environment — in other words, it is an external function. A use case diagram shows for one or more system functions which external entities may communicate with the system during an occurrence of each of these functions.
- The Shlaer/Mellor method uses a **communication diagram** to represent possible object communications [48]. This is a directed graph in which the nodes represent object classes or external entities and the edges represent possible communications. These are asynchronous in the Shlaer/Mellor method.

This exhausts the techniques used for the specification of communication. There are two other techniques, that can be used to *illustrate* the communication and behavior of a system.

- A **sequence diagram** consists of a set of vertical lines, each representing a communicating entity. The downwards direction represents the advance of time. Arrows between these lines represent communications. Sequence diagrams have been used for a long time in telecommunication systems, where they are standardized as message sequence charts [28]. Variations of the technique are used in object-oriented analysis [31, 46] to represent the communication between objects or between the system and its environment. In an attempt to standardize on an object-oriented version of the technique it is adopted by the UML [42]. A sequence diagram represents behavior as well as communication. However, it does not represent all possible behaviors and communications, but only those that can occur in a particular scenario.

	function 1	....	function n
component 1			
.....			
component m			

Figure 4: Format of a function decomposition table.

- **Collaboration diagrams** are directed graphs in which the nodes represent communicating entities and the edges communications. The edges are numbered to represent the sequence in which communications take place. Collaboration diagrams can be used as alternative to sequence diagrams. Like sequence diagrams, there are many versions, that differ in the elaborations and adornments that they allow. The technique plays a central role in responsibility-driven design [64] and in a number of object-oriented methods such as Booch [6], Fusion [9], and Syntropy [10]. It has been adopted in the UML as alternative to sequence charts [42].

The communication specification techniques are clearly compatible and can be integrated in an obvious way. For example, a context diagram can show where the events come from and where the responses of the system go to. If we add an STD to specify the behavior of external event-response pairs, a sequence or collaboration diagram can be used to illustrate possible communication sequences of the system generated by this behavior. In the specification of external interactions, therefore, there is no incompatibility between structured and object-oriented analysis. To find the difference, we must look at decomposition specification techniques.

3.2 Decomposition

Taking a systems engineering view, we can represent the allocation and flowdown of external functions to components by means of a **function decomposition table**, also called a traceability table in systems engineering [11, page 192]. [12]. In figure 4, the top row lists all external functions of the system (at a certain level of refinement) and the leftmost column represents all components of the system (at a certain level of aggregation). An entry of the table represents the functions that a component must have to in order to realize an external function. A column of the table represents all functions that act together to realize an external function. The table relates all perspectives that we identified: functions, decomposition, behavior (ordering of functions in a row) and communication (columns). Note that each such table corresponds to one point in the magic square of figure 2, i.e. a refinement level and an aggregation level.

3.2.1 Decomposition specification

Yourdon-style structured analysis uses **entity-relationship diagrams** (ERDs) and **data flow diagrams** (DFDs) to represent software decomposition. Figure 5 contains a simple DFD that represents three components and the way they communicate with each other

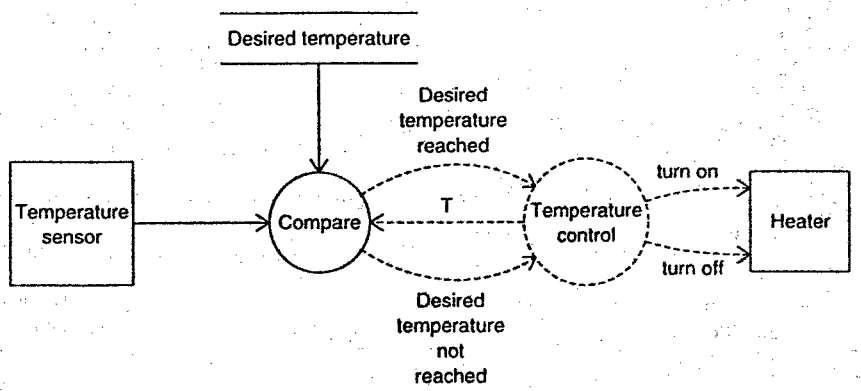


Figure 5: A simple data flow diagram.

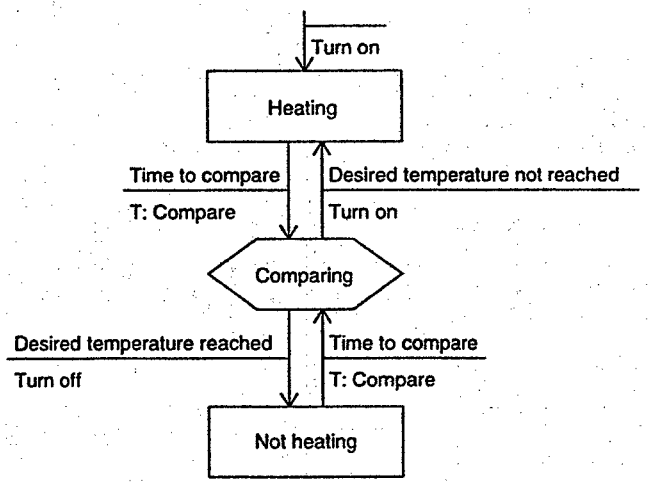


Figure 6: A Mealy STD for the Temperature control process with a decision state.

Figure 7: A simple entity-relationship diagram.

and with external entities. Temperature sensor and Heater are external entities used to control the temperature of a fluid in a cooking tank. These are not components of the system but part of the environment. The rest of the diagram illustrates that DFDs recognize three different kinds of system components:

- The **data store** Desired temperature represents the desired temperature of the fluid. A data store is a place where part of the state of the system can be stored.
- The **data process** Compare compares the value stored in Desired temperature with the value measured by Temperature sensor and sends the result to Temperature control. A data process is a function that transforms input data into output data.
- The **control process** Temperature control is a finite state machine that periodically triggers the data process Compare and, depending upon the answer, turns the Heater on or off.

Dashed arrows represent signals, solid arrows represent data flows. The Mealy STD for Temperature control is shown in figure 6. It shows that Temperature control triggers an external data process by the action T: Compare and then waits for the answer in the state Comparing. It moves to the Heating or Not heating state depending upon the answer, turning the heater on or off accordingly. Comparing is called a **decision state**.

The structure of all data in the system, stored or manipulated, can be represented by an ERD. Figure 7 shows a fragment of an ERD that describes the structure of some relevant data. It shows that the system must contain data about the Batch of juice to be heated, the Recipe according to which the batch must be heated, and the Cooking tank in which the batch must be heated. The arrow from Batch to Recipe means that there is exactly one Recipe for each Batch. The line between Batch and Cooking tank means that there is a many-many relationship between these. The Yourdon method does not prescribe the relationship between ERD and DFD other than that at least the structure of all stored data must be represented by the ERD.

Yourdon structured analysis thus recognizes three kinds of components to be listed in the leftmost column of the function decomposition table:

- data stores,
- data processes, and
- control processes.

Object-oriented methods recognize only one kind of component:

- objects, that encapsulate all these three aspects.

Each object contains data, can perform computations with these data, and has a behavior over time. This mans that object behavior can be specified by extended STDs. What this means can best be illustrated by an example.

Figure 8 shows a possible **class-relationship diagram** (CRD of an object-oriented decomposition of the control software. Each rectangle represents an object class. Temperature_control_S has two attributes, Desired_temp and Actual_temp. To simplify the diagram,

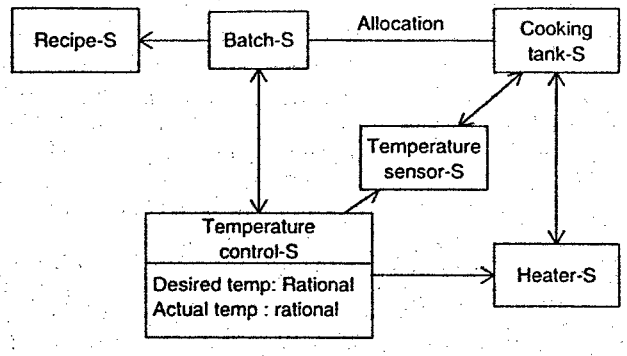


Figure 8: A simple class-relationship diagram.

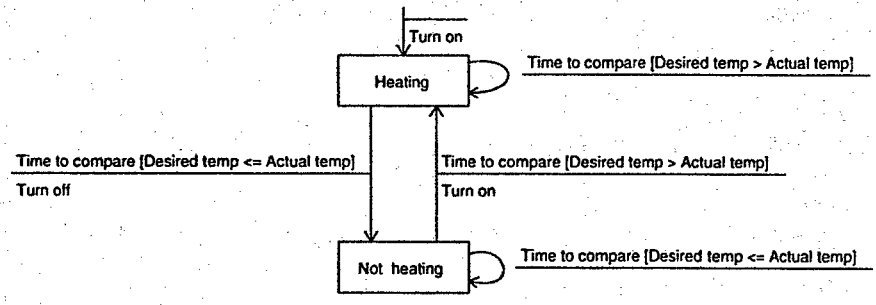


Figure 9: An extended Mealy STD for the Temperature control object.

we omitted the attributes of other objects. The diagram is an extension of the ERD with object classes that correspond to external entities and to a control process in the DFD. Bidirectional arrows in the diagram represent one-one relationships, single arrows represent many-one relationships. To emphasize that all objects represented by the CRD are software components, we added -S to their names. We did not need to do this in the ERD of figure 7, because in structured analysis, ERDs always represent data structures of software systems.

The behavior of the Temperature control object can be specified by an extended STD such as shown in figure 9. The STD uses the two attributes of Temperature control as local variables. They receive values from the environment of the Temperature control, in this case the Temperature sensor and Recipe software objects. This is not shown in the CRD: the lines in that diagram represent relationships between the software objects, which tells us which software objects know the identity of which other software objects. They do not tell us which communications between objects take place. The variables are tested in guards, which are denoted by Boolean expressions written between square brackets. Thus, if the event Time to compare occurs when the temperature control is in state Not heating. then the transition to Heating occurs if Desired temp > Actual temp, and the

heater is turned on.

Now, compare this with the Mealy STD in figure 6. It is clear that separating data and control, as is done in DFD models, makes for more complex models because this forces us to introduce decision states in the Mealy machine to await the outcome of decisions, and it forces us to introduce data processes that compute the decisions, and that communicate with the Mealy machine. The communication typically has the following structure: The Mealy machine triggers a data process that must compute a decision, and the data process sends its output as an event to the Mealy machine. Further complexity is introduced because DFDs separate data processing from data storage. This means that the data process does not have the data to compute the decision. It must get this from an input data flow and/or a data store. Data that survives a single external system transaction must be stored in data store, whereas in OO models this data persists in the state of objects. The consequence of separating data processing, data storage and control is that DFD-based models are considerably harder to understand than object-oriented models. Also, these separations are incompatible with the encapsulation principle of object-oriented decomposition. I conclude that DFDs are incompatible with object-oriented decomposition.

3.2.2 Decomposition heuristics

There are several criteria that can be used to find a decomposition. **Functional decomposition** uses external functionality as criterion. In its simplest form, every external elementary function corresponds to a component that implements that function. This would lead to a diagonal in the function decomposition table, mapping external functionality to internal structure. This is alright if it leads to small interfaces between components and if the external functionality never changes. If either of these conditions is false it is a bad decomposition. It is often claimed that functional decomposition is incompatible with object-orientation. However, there is nothing in the concept of an object that prevents us from using a functional decomposition criterion for an object-oriented decomposition. For example, in figure 8, Temperature control corresponds to a function of the system.[3]

A second decomposition criterion that can be used is **subject domain-oriented decomposition**. In section 2 it was noted that software systems interact with their environment by exchanging symbol occurrences with the environment. The subject domain of a software system was then defined as the part of the world referred to by these symbols. For example, the subject domain of a database system is the part of the world represented by the database, and the subject domain of a control system is the part of the world controlled by the system. In **subject domain-oriented partitioning**, there is a correspondence between the software system decomposition and a subject domain decomposition. For example, figure 8 contains software objects that correspond to a batch, a recipe and a cooking tank.

This decomposition principle has been adopted by many object-oriented methods but contrary to what is often thought, it is not characteristic of object-oriented methods. For example, the principle is central in JSD [29], which is not purely object-oriented. Furthermore, as I show elsewhere [55], the principle can easily be applied to data flow

[3] In JSD, such objects are called long-running functions [29] and in Objectory, they are called control objects [31].

modeling. Finally, we find improper use of the principle of subject-domain partitioning in object-oriented methods. For example, the Fusion method recommends making an ER-like domain model first and then drawing a boundary around the object classes that will constitute the system [9]. By the act of drawing this boundary, objects in the domain (outside the system) become objects inside the system. Ojects outside the boundary remain subject domain entities. This confuses the subject domain with the internal decomposition of the system. The resulting diagram combines features of a CRD showing the essential decomposition of the system with features of a context diagram showing the communications with the environment. Published examples show models of the domain (outside the system) that already contain system functions and other essential system components [2]. In some cases, the domain model contains two copies of an entity, one for the external entity and one for a essential system component. Use of the method in teaching show difficulties precisely at this point [15].

In addition to functional decomposition and subject-domain-oriented decomposition, there are two intermediary decomposition heuristics. **Event partitioning** recommends defining one system component for every elementary event-response pair [37] and **device partitioning** recommends defining one system component for every external [66. pages 355, 509]. In practice, a combination of these heuristics will be used. For example, figure 8 contains components that correspond to external devices (Heater-S and Temperature sensor-S) and we have seen that it also contains components that correspond to subject domain objects and to a system function.

4 The Techniques in TRADE

Figure 10 lists the techniques that have been adopted in TRADE. The reason for adopting these techniques is that together, they the kinds of properties that can be specified in a wide variety of structured and object-oriented methods [57]. For each dimension of our framework, we chose simple and useful techniques that can be used to represent system properties along that dimension. Because the primary aim of TRADE is to teach informatics students about software specification techniques, ease of understanding is preferred above expressive power. This is the reason why extended Mealy machines rather than extended statecharts are chosen to represent behavior. Experience has taught that students tend to produce unnecessarily complex models when using statecharts to represent behavior. Also, local variables gives us all the added expressive power that we need: state nesting and parallelism can be expressed by a set of communicating Mealy machines. And as argued earlier, the introduction of local variables in any STD technique allows us to avoid the complexities of data flow diagrams.

Simplicity of techniques also makes it feasible to define the connections that must hold between different parts of a coherent specifications in an understandable manner. Without going to details, if the techniques are used as indicated in figure 11, the links shown in figure 12 must hold. The entries of the table indicate the techniques between which there exist links. The table is symmetric around the diagonal, which is why only half of it is shown. A brief explanation of the entries of the table follows, where the entries are identified by a pair (row.column).

(2.1) The root of the function refinement tree is labeled by the mission statement.

	Function specification technique	Behavior specification technique	Communication specification technique	Decomposition specification technique
Mission statement	X			
Function refinement tree	X			
Event-response specification	X			
Pre-postcondition specification	X			
Extended Mealy machine diagram		X		
Communication diagram			X	
Sequence diagram		X	X	
Class diagram				X
Function decomposition table				X

Figure 10: The techniques in TRADE.

	Subject domain	External system interactions	System decomposition	Component interactions
Mission statement		Mission		
Function refinement tree		Mission refinement		
Event-response specification		Elementary external functions		Component functions
Pre-postcondition specification		Elementary external functions		Component functions
Extended Mealy machine diagram	Subject domain entity behavior			Component behavior
Communication diagram		Context diagram Use case diagram		Component interaction
Sequence diagram		External interaction sequences		Component interaction sequences
Class diagram	Subject domain decomposition		Essential decomposition	
Function decomposition table			Allocation and flowdown of functions	

Figure 11: Use of the techniques in TRADE.

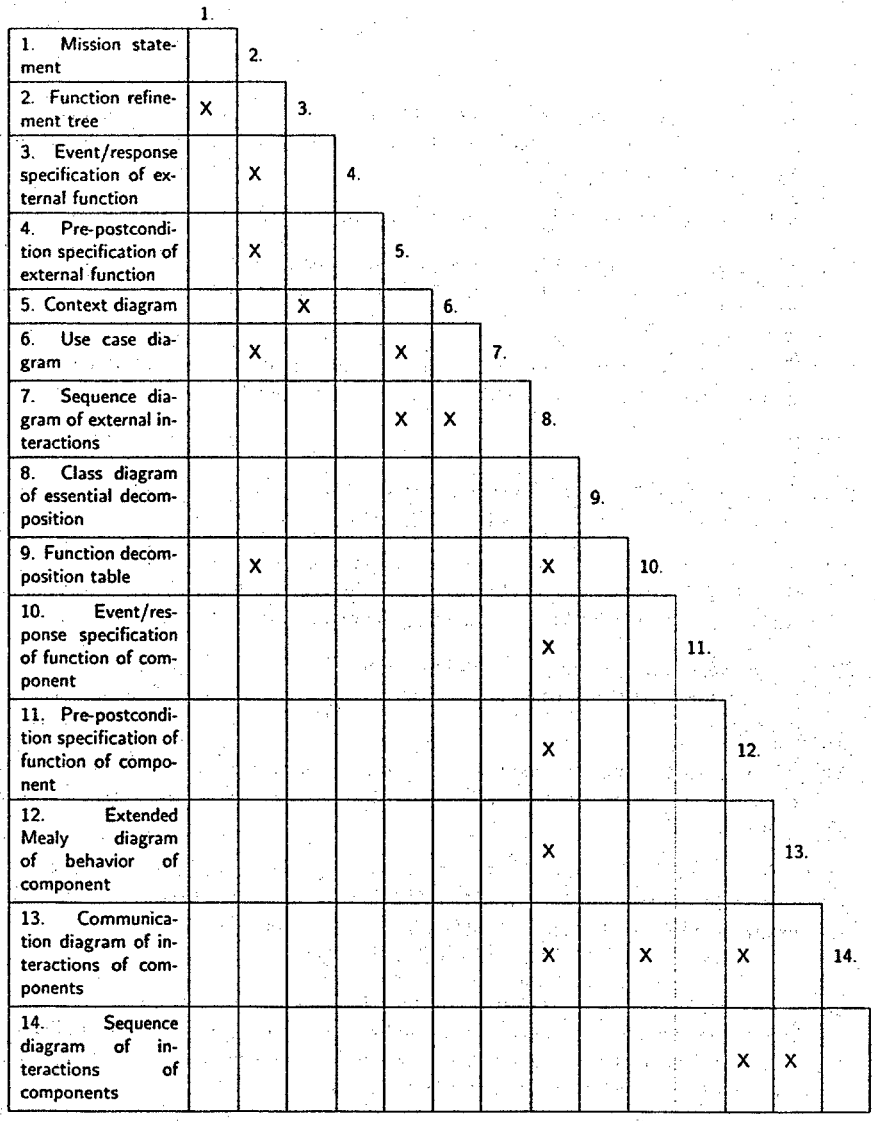


Figure 12: Links between parts of a coherent specification.

(3.2) Each leaf of the function refinement tree represents an external function, specified by means of an event-response specification and/or a pre-postcondition specification. Each external function specification corresponds to a node in the function refinement tree.

(4.2) See (3.2).

(5.3) The event sources and response destinations of an event-response specification of an external function are external entities represented by the context diagram, that interact with the system.

(6.2) Each use case corresponds to a node in the function refinement tree.

(6.5) The interactions between a use case and an external entity, represented in a use case diagram, also occur in the context diagram between the system and an external entity.

(7.5) The external communications in a sequence diagram of external interactions correspond to the external communications in the context and in use case diagrams.

(7.6) See (7.5).

(9.2) The top row of the function decomposition table corresponds one-one to the leaves of the function refinement tree (they represent external functions).

(9.8) The leftmost column of the function decomposition table corresponds one-one with the set of classes in the class diagram. In addition, the entries of the table must be consistent with the interface of the components declared in the class diagram.

(10.8) The events and responses in a event-response specification of a component must be consistent with the events and responses of the component declared in the class diagram.

(11.8) The terms in a pre-postcondition specification of an interaction of a component must be consistent with the attributes of the component declared in the class diagram.

(12.8) The events and responses of the transitions in the Mealy diagram of a class must correspond with the events and responses of the class declared in the class diagram. In addition, the local variables used in the Mealy diagram correspond to the attributes declared for the class in the class diagram.

(13.8) The communications represented in a communication diagram of component interactions correspond with the events and responses declared in the class diagram. (Each of these is part of the interface of the object and consists of a communication with another object or with an external entity.)

(13.10) See (13.8).

(13.12) See (13.8).

(14.12) The sequence of communications in a sequence diagram of component communications is consistent with the Mealy diagram of the communicating components.

(14.13) The communications in a sequence diagram of component communications are consistent with the communications represented by the communication diagram.

This list suffices to give an impression of the connection rules. There are two ways to make this more precise, by means of formalization and by means of a metamodel. To formalize the diagrams and their links, a formalization based upon order-sorted dynamic logic and process algebra will be used [53, 63, 62]. This is particularly important for the rules in the above list that contain the word "consistent". Definition of a metamodel is ongoing work, which is part of the specification and implementation of the TCM software tool [59].

Figure 12 does not define links for the subject domain model (a class diagram with extended Mealy machines). These parallel the links between the class diagram and Mealy machines for the system decomposition. If subject-domain-oriented decomposition for the system is used, then there will be links between the class diagrams of the subject domain and that of the system. These are however a result of design decisions and are not a consequence of the semantics of the notations. Figure 12 only lists the links that must hold in all cases.

Observe that we can define the links only because we presuppose our framework for software design techniques. It is this framework that gives us the concepts of subject domain, functions, behavior, communication and decomposition, which give the links meaning and that allows is to justify that the links must be present.

Comparing the techniques in TRADE with the UML [43], we observe that the structure of TRADE models corresponds with that of UML models in that a software system is viewed as a collection of interacting objects, whose structure is represented by a class diagram and whose behavior is represented by state transition diagrams. In addition, a TRADE model represents external functionality by a mission statement and a function refinement tree, and adds traceability by defining a function decomposition table. TRADE only uses the simplest possible state machine notation (Mealy machines) rather than the complex statechart notation, and omits collaboration diagrams, which have roughly the same expressive power as sequence diagrams.

Just like the Yourdon Systems Method [66], TRADE models contain an elaborate specification of external functionality, using roughly the same techniques as used in YSM. Unlike YSM, the essential decomposition is not represented by means of data flow models but by means of a class diagram.

Another interesting comparison to make is with SDL [5], used for modeling telecommunication systems. An SDL model represents a system as a hierarchy of subsystems, called *blocks*, that may communicate via channels. Each block at the bottom of the hierarchy consists of one or more communicating processes, each of which is specified by means of an extended finite state machine. The major difference with TRADE models is that TRADE contains more techniques for specifying external functionality but contains no technique for representing subsystems. Addition of a subsystem representation technique, and heuristics for partitioning a system into subsystems, is a topic of current research.

Turning to the heuristics that can be used to apply the techniques in TRADE, these have already been described in section 3.2.2. These heuristics have their source in struc-

tured and object-oriented methods and can all be used in combination with the techniques in TRADE.

5 Discussion and Conclusions

TRADE techniques can be used in different design strategies, ranging from waterfall to incremental or evolutionary. They look familiar to developers with a structured background as well as those with an object-oriented background and therefore should help in combining the best elements of both practices. The essential element in this is to institute a systems engineering way of working, in which specification of external interactions is separated from a specification of internal decomposition, and explicit traceability from external interactions to internal components is maintained. Structured techniques for external interaction specification can then be seamlessly combined with object-oriented techniques for essential decomposition. DFDs cannot be integrated this way and should be dropped. However, I argued that functional decomposition is compatible with object-oriented decomposition. It can also be combined with other decomposition criteria, such as device partitioning and subject-domain-oriented partitioning.

To validate the TRADE framework, it has been applied to the industrial production cell case [54] and in the Esprit project 2RARE to the specification of a system for video on demand [61]. Two other case studies are available on the web [50, 58], and several others are in preparation. Further validation will take place in teaching, where it will be used to teach techniques in a method-independent way. Use of the TRADE framework in teaching is supported by a graphical editor called TCM (Toolkit for Conceptual Modeling), freely available for teaching and research purposes [13]. It supports most of the techniques discussed in this paper. Validation of another kind takes place by providing a formal semantics to the techniques in TRADE. A formal semantics of a combination of objects with behavior and communication, based on order-sorted dynamic logic and process algebra, has been given earlier [53, 63, 62]. Current work concentrates on declarative and operational semantics of behavior specifications so as to provide an execution semantics for STDs in TCM [59]. The methodological role of this is to strengthen the tools in TRADE by making their meaning and interconnections explicit. Our hope is that this makes the tools easier to use without burdening the tool user with the formal foundations.

References

[1] B. Alabiso. Transformation of data flow analysis models to object oriented design. In N. Meyrowitz, editor. *Object-Oriented Programming Systems, Languages and Applications. Conference Proceedings.* pages 335–353. ACM Press, 1988. SIGPLAN Notices, volume 23.

[2] M. Awad, J. Kuusela, and J. Ziegler. *Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion.* Prentice-Hall. 1996.

[3] S.C. Bailin. An object-oriented requirements specification method. *Communications of the ACM.* 32:608–623, 1989.

[4] M. von der Beeck. A comparison of Statecharts variants. In H. Langmaack, W.P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128–148. Springer, 1994. Lecture Notes in Computer Science 863.

[5] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from protocol Specification.* Prentice-Hall, 1991.

[6] G. Booch. *Object-Oriented Design with Applications. Second edition.* Benjamin/Cummings, 1994.

[7] R. Bræk and Ø. Haugen. *Engineering Real-Time Systems.* Prentice-Hall, 1993.

[8] D. Coleman. Fusion with use cases: Extending Fusion for requirements modeling. Available through URL: http://www.hpl.hp.com/fusion/index.html, 1996.

[9] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The FUSION Method.* Prentice-Hall, 1994.

[10] S. Cook and J. Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy.* Prentice-Hall, 1994.

[11] A.M. Davis. *Software Requirements: Objects, Functions, States.* Prentice-Hall, 1993.

[12] Defense Systems Management College. Systems engineering management guide. Technical report, Technical Managament Department, January 1990.

[13] F. Dehne and R.J. Wieringa. Toolkit for Conceptual Modeling (TCM): User's Guide. Technical Report IR-401, Faculty of Mathematics and Computer Science, *Vrije Universiteit*, De Boelelaan 1081a, 1081 HV Amsterdam, 1996. http://www.cs.vu.nl/~tcm.

[14] T. DeMarco. *Structured Analysis and System Specification.* Yourdon Press/Prentice-Hall, 1978.

[15] G. Eckert. *Improving the analysis stage of the Fusion method*, pages 276–313. Prentice-Hall, 1996.

[16] R.B. France. Semantically extended data flow diagrams: a formal specification tool. *IEEE Transactions on Software Engineering*, 18(4):329–346, April 1992.

[17] A.D. Hall. *A Methodology for Systems Engineering.* Van Nostrand, 1962.

[18] A.D. Hall. Three-dimensional morphology of systems engineering. *IEEE Transactions on System Science and Cybernetics*, SSC-5(2):156–160, 1969.

[19] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987. Preliminary version appeared as Technical Report CS 84-05, The Weizmann Institute of Science, Rehovot, Israel, February 1984.

[20] D. Harel and E. Gery. Executable object modeling with statecharts. In *Proceedings of the 18th International Conference on Software Engineering*, pages 246–257. IEEE Press, 1996.

[21] D. Harel and E. Gery. Executable object modeling with statecharts. *Computer*, 30(7):31–42, July 1997.

[22] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: a working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16:403–414, April 1990.

[23] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[24] D. Harel and A. Pnueli. On the development of reactive systems. In K. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer, 1985. NATO ASI Series.

[25] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings. Symposium on Logic in Computer Science*, pages 54–64. Computer Science Press, June 22–25 1987.

[26] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[27] i-Logix. The Languages of STATEMATE. Technical report, i-Logix Inc., 22 Third Avenue, Burlington, Mass. 01803, U.S.A., January 1991. To be published as D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The STATEMATE Approach.*

[28] ITU. *Criteria for the Use and Applicability of Formal Description Techniques: Message Sequence Charts (MSC).* International Telecommunication Union, 1994. Z.120 (03/93).

[29] M. Jackson. *System Development.* Prentice-Hall, 1983.

[30] M. Jackson. *Software Requirements and Specifications: A lexicon of practice, principles and prejudices.* Addison-Wesley, 1995.

[31] I. Jacobson, M. Christerson, P. Johnsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Prentice-Hall, 1992.

[32] P. Jalote. Functional refinement and nested objects for object-oriented design. *IEEE Transactions on Software Engineering*, 15(3):264-270, March 1989.

[33] I. Lakatos. *Proofs and Refutations.* Cambridge University Press, 1976. Edited by J. Worall and E. Zahar.

[34] J. Martin. *Information Engineering, Book I: Introduction.* Prentice-Hall, 1989.

[35] J. Martin. *Information Engineering, Book II: Planning and analysis.* Prentice-Hall, 1989.

[36] J. Martin and J. Odell. *Object-Oriented Methods: A Foundation.* Prentice-Hall, 1995.

[37] S.M. McMenamin and J.F. Palmer. *Essential Systems Analysis.* Yourdon Press/Prentice Hall, 1984.

[38] R. Milner. *A Calculus of Communicating Systems.* Springer, 1980. Lecture Notes in Computer Science 92.

[39] B. Møller-Pedersen, D. Belsnes, and H.P. Dahle. Rationale and tutorial on OSDL: an object-oriented extension of SDL. *Computer Networks and ISDN Systems*, 13:97–117, 1987.

[40] Objectory AB. *Objectory: Requirements Analysis, Version 3.6*, 1995.

[41] D.L. Parnas and P.C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12:251-257, 1986.

[42] Rational. *Unified Modeling Language: Notation Guide, Version 1.0.* Rational Software Corporation, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951. 13 january 1997. URL http://www.rational.com/ot/uml.html.

[43] Rational. *Unified Modeling Language: Notation Guide, Version 1.1.* Rational Software Corporation, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951. 1 September 1997. URL http://www.rational.com/uml/1.1/.

[44] Rational. *Unified Modeling Language: Semantics, Version 1.1.* Rational Software Corporation, 1 September 1997. URL http://www.rational.com/uml/1.1/.

[45] N.F.M. Roozenburg and J. Eekels. *Product design: Fundamentals and Methods.* Wiley, 1995.

[46] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented modeling and design.* Prentice-Hall, 1991.

[47] E. Seidewitz and M. Stark. Toward a general object-oriented software development methodology. *ADA Letters*, 7(4):54–67, july/august 1987.

[48] S. Shlaer and S.J. Mellor. *Object Lifecycles: Modeling the World in States.* Prentice-Hall, 1992.

[49] W. Swartout and R. Balzer. On the inevitable intertwining of specification and implementation. *Communications of the ACM*, 25:438–440, 1982.

[50] B. van Vlijmen and R.J. Wieringa. Using the Tools in TRADE, I: A Decision Support System for Traffic Light maintenance. Technical Report IR-435, Faculty of Mathematics and Computer Science, *Vrije Universiteit*, November 1997. ftp://ftp.cs.vu.nl/pub/roelw/97-TRADE01.ps.Z.

[51] P.T. Ward. How to integrate object orientation with structured analysis and design. *Computer*, pages 74–82, March 1989.

[52] P.T. Ward and S.J. Mellor. *Structured Development for Real-Time Systems.* Prentice-Hall/Yourdon Press, 1985. Three volumes.

[53] R.J. Wieringa. A formalization of objects using equational dynamic logic. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *2nd International Conference on Deductive and Object-Oriented Databases (DOOD'91)*, pages 431–452. Springer, 1991. Lecture Notes in Computer Science 566.

[54] R.J. Wieringa. LCM 3.0: Specification of a control system using dynamic logic and process algebra. In C. Lewerentz and T. Lindner, editors, *Formal Development of Reactive Systems — Case Study Production Cell*, pages 333–355. Springer, 1994. Lecture Notes in Computer Science 891.

[55] R.J. Wieringa. Combining static and dynamic modeling methods: a comparison of four methods. *The Computer Journal*, 38(1):17–30, 1995.

[56] R.J. Wieringa. *Requirements Engineering: Frameworks for Understanding.* Wiley, 1996.

[57] R.J. Wieringa. A survey of structured and object-oriented software specification methods and techniques. Technical report, Faculty of Mathematics and Computer Science, *Vrije Universiteit*, De Boelelaan 1081a, 1081 HV Amsterdam, 1997. To be published, *ACM Computing Surveys.*

[58] R.J. Wieringa. Using the tools in TRADE, II: Specification and design of a meeting scheduler system. Technical Report IR-436, Faculty of Mathematics and Computer Science. *Vrije Universiteit*, November 1997. ftp://ftp.cs.vu.nl/pub/roelw/97-TRADE02.ps.Z.

[59] R.J. Wieringa and J. Broersen. A minimal transition system semantics for lightweight class- and behavior diagrams. Technical report, Faculty of Mathematics and Computer Science, *Vrije Universiteit*, De Boelelaan 1081a, 1081 HV Amsterdam, December 1997. Submitted for publication.

[60] R.J. Wieringa and E. Dubois. Integrating semi-formal and formal software specification techniques. Technical report, Faculty of Mathematics and Computer Science, *Vrije Universiteit*, De Boelelaan 1081a, 1081 HV Amsterdam, 1997. To be published, *Information Systems.*

[61] R.J. Wieringa, E. Dubois, and S. Huyts. Integrating semi-formal and formal requirements. In A. Olivé and J.A. Pastor, editors, *Advanced Information Systems Engineering*, pages 19–32. Springer, 1997. Lecture Notes in Computer Science 1250.

[62] R.J. Wieringa, W. de Jonge, and P.A. Spruit. Using dynamic classes and role classes to model object migration. *Theory and Practice of Object Systems*, 1(1):61–83, 1995.

[63] R.J. Wieringa and J.-J.Ch. Meyer. Actors, actions, and initiative in normative system specification. *Annals of Mathematics and Artificial Intelligence*, 7:289–346, 1993.

[64] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software.* Prentice-Hall, 1990.

[65] E. Yourdon. *Modern Structured Analysis.* Prentice-Hall, 1989.

[66] Yourdon Inc. *Yourdon™ Systems Method: Model-Driven Systems Development.* Prentice-Hall, 1993.

Index of Authors