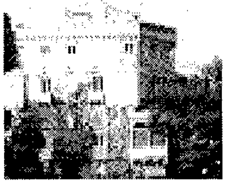
Proceedings of the

2000 Monterey Workshop on Modelling Software System Structures in a fastly moving scenario

Santa Margherita Ligure, Italy
June 13-16, 2000

Co-sponsored by
ARO, NSF, ARL (European Research Office),
INDAM (GNIM), University of Genoa

Hosted by
DISI - University of Genoa, Italy



Modelling Software System Structures in a fastly moving scenario

June 13-16, 2000
Santa Margherita Ligure, Italy

Workshop Programme and Schedule

Monday June 12

17.00-18.00 Registration
18.00-19.30 Welcome Reception & Registration
19.30 Dinner

Tuesday June 13

9.00 Opening

9.15-10.00 Chair E. Astesiano
Manfred Broy Dynamics, Mobility, and System Topology of Hardware/Software Nets: Towards a Mathematical Model

10.00-10.30 Coffee break

10.30-12.00 Chair T. Maibaum
Cliff Jones Formal Methods and Dependability
Dave Robertson Experimental Analysis for Large Agent Systems
12.00-12.30 Discussion

12.30-15.00 Lunch break

15.00-16.30 Chair M. Bidoit
Purushothaman Iyer Unfoldings of Infinite State Systems
Aloysius K. Mok Tracking Real-Time Systems Requirements

16.30-16.45 Coffee break

16.45-19.00 Chair C. Montangero
Mikhail Auguston Lightweight Semantics Models for Program Testing and Debugging Automation
Paola Inverardi Software Architectures and Component Programming
Marco Bernardo Performance Evaluation of Architectural Types: A Process Algebraic Approach

19.30 Dinner

Wednesday June 14

8.30-10.00 Chair Luqi
Jeffrey Tsai Compositional Approach for Modeling and Verification of Component-Based Software Systems
Mike Reed Automated Formal Support for Component-Based Systems: A Commercial Strategy

10.00-10.30 Coffee break

10.30-12.00 Chair P. Inverardi
Alex Wolf Middleware Component Frameworks: A Challenge for Software Architecture Research
Heinrich Hussmann Towards Practical Support for Component-Based Software Development Based on Formal Specification
Carlo Montangero Specification and Composition of Software Components: Formal Methods Meet Standards

16.45-20.01 Discussion on CBSE (all speakers; A.Wolf chair)

12.30-15.00 Lunch break

15.00-16.30 Chair M. Cerioli
Christine Choppy Using CASL to Specify the Requirements and the Design: A Problem Specific Approach
Oleg Sokolsky Comparative Analysis of Design Alternatives in Embedded Systems

16.30-16.45 Coffee break

16.45-20.00 Free time for a visit to Portofino

20.30 Banquet

Modelling Software System Structures in a fastly moving scenario

June 13-16, 2000
Santa Margherita Ligure, Italy

Workshop Programme and Schedule

Thursday June 15

8.30-10.15 Chair P. Iyer

Insup Lee Run-time Monitoring and Steering based on Formal Specifications

Nikolaj Bjoerner How to Write your Specification, Synthesize your Program and Execute it too

Doug Lange A Formal Model of System and Software Engineering Experience

10.15-10.30 **Coffee break**

10.30-12.45 Chair D. Bjørner

Connie Heitmeyer Formal Analysis of Software Requirements: Integrating Different Techniques

Zohar Manna Verification Diagrams: Logic + Automata

Dan Berry Appliances and Software: The Importance of the Buyer's Warranty and the Developer's Liability in Promoting the Use of Formal Methods

12.45-15.00 **Lunch break**

15.00-16.45 Chair J.L. Fiadeiro

Dines Bjørner Domain Engineering - "Upstream" from Requirements Engineering

Ar*mando Haeberer* An Alternative Approach for Specification-Based Functional Verification Testing

Valdis Berzins Static Analysis for Program Generation Templates

16.45-17.00 **Coffee break**

17.00-18.10 Chair M. Broy

Gianna Reggio JTN: A Java Targeted Formal Visual Notation for the Design of Reactive Systems

Werner Damm Breathing Life into Message Sequence Charts

18.10-19.00

Panel/discussion

Emerging Technologies in the Practice: UML and the Like (Broy chair, Damm, Ciancarini, Reggio, Reed)

19.30 **Dinner**

Friday June 16

8.30-10.00 Chair H. Hußmann

Maritta Heisel Toward an Evolutionary Software Technology

Josè Fiadeiro Coordination: the Evolutionary Dimension

10.00-10.30 **Coffee break**

10.30-11.55 Chair D. Berry

Du Zhang Applying Machine Learning Algorithms in Software Development

11.55-13.00

Final discussion & Closing

(Luqi, Broy, Zavada, Astesiano)

13.00 **Lunch**

List of Participants

Egidio	Astesiano	University of Genova , Italy
Mikhail	Auguston	New Mexico State University, USA
Marco	Bernardo	University of Torino, Italy
Daniel	Berry	University of Waterloo, Canada
Valdis	Berzins	Monterey Naval Postgraduate School, USA
Michel	Bidoit	Ecole Normale Supérieure de Cachan, France
Nikolaj	Bjørner	Kestrel Institute, USA
Dines	Bjørner	Technical University of Denmark, Denmark
Manfred	Broy	Technical University of Munich, Germany
Maura	Cerioli	University of Genova, Italy
Christine	Choppy	Université Paris Nord, France
Paolo	Ciancarini	University of Bologna, Italy
Werner	Damm	Univ. of Oldenburg, Germany
José L.	Fiadeiro	University of Lisbon, Portugal
Nicolas	Guelfi	Luxembourg Univ. of Appl. Science, Luxenbourg
Armando	Haeberer	Oblog Software SA, Portugal
Maritta	Heisel	University of Magdeburg, Germany
Constance	Heitmeyer	Naval Research Laboratory, USA
Heinrich	Hußmann	Dresden University of Technology, Germany
Paola	Inverardi	University of l'Aquila, Italy
Purushothaman	Iyer	North Carolina State University, USA
Cliff	Jones	University of Newcastle, UK
Doug	Lange	SPAWAR, USA
Insup	Lee	University of Pennsylvania, USA
Luqi		Monterey Naval Postgraduate School, USA
Tom	Maibaum	King's College London, UK
Zohar	Manna	Stanford University, USA
Aloysius	Mok	University of Texas, USA
Carlo	Montangero	University of Pisa, Italy
Anna	Philippou	University of Cyprus, Cyprus
Mike	Reed	Oxford University, UK
Gianna	Reggio	University of Genova, Italy
David	Robertson	University of Edinburgh, UK
Oleg	Sokolsky	University of Pennsylvania, USA
Jeffrey	Tsai	University of Illinois, USA
Alex	Wolf	University of Colorado, USA
John	Zavada	ARO/ERO - US Army, USA
Du	Zhang	California State University, USA

Workshop Chairs

Egidio Astesiano, Manfred Broy and Luqi

Programme Committee

Egidio Astesiano

Manfred Broy

Luqi

Carlo Ghezzi

Zohar Manna

Steering Committee

David Hislop

Frank Anger

Valdis Berzins

Purushothaman Iyer

Organization Committee

Gianna Reggio (Chair)

Maura Cerioli

Egidio Astesiano

Acknowledgement

The organizers of this workshop would like to thank the sponsors of the workshop: Army Research Office (ARO), National Science Foundation (NSF), Army Research Laboratory (ARL European Research Office), INDAM (GNIM) and the University of Genoa.

List of Participants

Egidio	Astesiano	University of Genova , Italy
Mikhail	Auguston	New Mexico State University, USA
Marco	Bernardo	University of Torino, Italy
Daniel	Berry	University of Waterloo, Canada
Valdis	Berzins	Monterey Naval Postgraduate School, USA
Michel	Bidoit	Ecole Normale Supérieure de Cachan, France
Nikolaj	Bjørner	Kestrel Institute, USA
Dines	Bjørner	Technical University of Denmark, Denmark
Manfred	Broy	Technical University of Munich, Germany
Maura	Cerioli	University of Genova, Italy
Christine	Choppy	Université Paris Nord, France
Paolo	Ciancarini	University of Bologna, Italy
Werner	Damm	Univ. of Oldenburg, Germany
José L.	Fiadeiro	University of Lisbon, Portugal
Nicolas	Guelfi	Luxembourg Univ. of Appl. Science, Luxenbourg
Armando	Haeberer	Oblog Software SA, Portugal
Maritta	Heisel	University of Magdeburg, Germany
Constance	Heitmeyer	Naval Research Laboratory, USA
Heinrich	Hußmann	Dresden University of Technology, Germany
Paola	Inverardi	University of l'Aquila, Italy
Purushothaman	Iyer	North Carolina State University, USA
Cliff	Jones	University of Newcastle, UK
Doug	Lange	SPAWAR, USA
Insup	Lee	University of Pennsylvania, USA
Luqi		Monterey Naval Postgraduate School, USA
Tom	Maibaum	King's College London, UK
Zohar	Manna	Stanford University, USA
Aloysius	Mok	University of Texas, USA
Carlo	Montangero	University of Pisa, Italy
Anna	Philippou	University of Cyprus, Cyprus
Mike	Reed	Oxford University, UK
Gianna	Reggio	University of Genova, Italy
David	Robertson	University of Edinburgh, UK
Oleg	Sokolsky	University of Pennsylvania, USA
Jeffrey	Tsai	University of Illinois, USA
Alex	Wolf	University of Colorado, USA
John	Zavada	ARO/ERO - US Army, USA
Du	Zhang	California State University, USA

Table of Contents

Foreword

The Workshop on Modelling Software System Structures in a fastly moving scenario was sponsored by the Army Research Laboratory European Research Office, U.S. Army Research Office, National Science Foundation, Istituto Nazionale di Alta Matematica/GNIM and Università di Genova.

This workshop is the 7th in a series of Software Engineering workshops, called "Monterey Workshops" from the Monterey Naval Postgraduate School, where they originated under the initiative of prof. Luqi.

The general aim of these workshops is formulating and advancing software engineering models and techniques, with the fundamental theme of increasing the practical impact of formal methods. Previous workshops have been devoted to "Real-time & Concurrent Systems", "Software Merging and Slicing", "Software Evolution", "Software Architecture", "Requirements Targeting Software", and "Engineering Automation for Computer Based Systems".

A major goal for this series of workshops is to help to focus the software engineering community on issues that are vital to improving the state of software engineering practice, bringing together American and European leading scientists actively engaged in the area.

The context for the workshop initiative is nicely set up in the words from the PITAC (the USA President's Information Technology Advisory Committee) 1998 Interim Report.

> *"The demand for software has grown far faster than the resources we have to produce it. The result is that desperately needed software is not being developed.*
>
> *Furthermore, the nation needs software that is far more usable, reliable, and powerful than what is being produced today."*
>
> *"...it has become clear that the processes of developing, testing, and maintaining software must change. We need scientifically sound approaches to software development that will enable meaningful and practical testing for consistency of specifications and implementations."*
>
> *Unfortunately, as the same interim report emphasizes, "current support (for research) is taking a short-term focus, looking for immediate returns, rather than investigating high-risk long-term technologies".*

As a consequence, there is a danger of even widening the gap between fundamental research and current (not always best-) practice. Indeed, together with long standing problems, such as the quest for software reliability, we are facing the need and partly the emergence of radically different ways of producing software.

The 7th Workshop, continuing the effort to bring together pragmatic and foundational research in software engineering, has primarily focused the attention on the major issues characterizing the new and rapidly evolving scenario of software development, such as the emphasis on high-level architectural aspects and the component-based and web-based software development.

Together with proposing new concepts and techniques, another major achievement of the workshop has been the demonstration that the wealth of past foundational research in SE can be uplifted to handle some, if not all, of the new problems posed, among others, by the different level of component and system granularity, the heterogeneity of components, the use of distribution and communication and the request for appropriate human-interface support.

The participation was well balanced, considering that the event took place in Italy: we had 38 participants, 15 from USA, 15 from Europe, 7 from Italy (including 3 local people) and 1 from Canada.

Altogether there have been 29 talks and two panels, each with five participants.

There has been a nice mix of technical talks and talks surveying/proposing hot topics. The discussion was quite alive and reached high peaks, especially in the discussions centered around Component based SE and the emergence of UML.

To mark the importance of the event, on Wednesday, at the official banquet, we have been honored by the presence of the President of the University of Genova, who welcomed us, also reacting positively to a nice dinner speech by Manfred Broy, who presented the motivation for the Workshop within the worldwide fastly moving scenario shaped by the IT explosion.

Dr. John Zavada also spoke at the banquet presenting the goals of the research support provided by his office and expressing the opinion that this kind of meeting USA/EUROPE should be more frequent, because they offer the opportunity of merging different cultures. The difference in cultures was indeed clearly visible at the workshop, which however was already showing some remarkable convergence in attitude.

I think it is fair to summarize the overall feeling, saying that, as result of the workshop, everybody really got a picture of a fastly moving scenario and the many problems we have to face rapidly to cope with the pace in software development, as it was summarized by Luqi and Manfred Broy in the closing session and discussion and in the words of a postworkshop message by Dr John Zavada (ARL/ERO): "I enjoyed the workshop and the discussions that we had. I think that I now have a better understanding of the issues facing software development."

Egidio Astesiano

DISI - University of Genova
Via Dodecaneso, 35, 16146 Genova
ITALY

SAT-solving the Coverability Problem for Unbounded Petri Nets

Parosh Aziz Abdulla
Dept of Computer Systems
Uppsala University
Uppsala, Sweden
parosh@docs.uu.se

S. Purushothaman Iyer*
Dept of Computer Science
North Carolina State University
Raleigh, NC 27695-7534
purush@csc.ncsu.edu

Aletta Nylén
Dept of Computer Systems
Uppsala University
Uppsala, Sweden
aletta@docs.uu.se

Abstract

Verifying systems using net unfoldings to represent the state space is a process containing two steps: generation of the unfolding and reasoning about the unfolding. In a recent paper [AIN00] we generalized the notion of unfoldings to unbounded Petri nets and showed how to capture a symbolic representation of the state space of these nets, thus completing the first of the two steps discussed above. We now continue with our experimentation and show how to reason with the unfolding obtained. In particular, we show how to use a SAT-solver to solve the coverability problem for unbounded Petri nets. The results of our experiments show that the use of unfoldings, in spite of the two-step process, has better time and space characteristics than an implementation that does not use unfoldings. In effect, we provide the first evidence for the conjecture that the two step process based on unfoldings could be better than a single step verification process that considers all interleavings.

1 Introduction

Model checking has had a great impact as an efficient method for algorithmic verification of finite-state systems. A limiting factor in its application is the *state space explosion* problem, which occurs since the number of states grows exponentially with the number of components in the system. Therefore, much effort has been spent on developing techniques for reducing the effect of state space explosion in practical applications. One such a technique is that of *partial orders* which is based on the observation that not all interleavings of a given set of independent actions need to be explored during model checking. Several criteria for independency has been given, e.g., *stubborn sets* [Val90], *persistent sets* [GW93] or *ample sets* [Pel93]. A method which has drawn considerable attention recently is that of *unfoldings* [McM95, ERV96, ER99]. Unfoldings are *occurrence nets*: unrollings of Petri nets that preserve their semantics. Although unfoldings are usually infinite, it is observed in [McM95] that we can always construct a finite initial prefix of the unfolding which captures its entire behavior, and which in many cases is much smaller than the state space of the system. Unfoldings have been applied to n-safe (i.e., finite-state) Petri nets, and more recently to other classes of finite-state systems such as synchronous products of finite transition systems [LB99, ER99].

There has also been numerous efforts to extend the applicability of model checking to the domain of infinite-state systems. This has resulted in several highly nontrivial algorithms for verification of timed automata,

*Supported in part by US ARO under grant P-38682-MA and by STINT

lossy channel systems, (unbounded) Petri nets, broadcast protocols, relational automata, parameterized systems, etc. These methods operate on symbolic representations, called *constraints* each of which may represent an infinite set of states. However, in a manner similar to finite-state verification, many of these algorithms suffer from a *constraint explosion* problem limiting their efficiency in practical applications. As the interest in the area of infinite-state systems increases, it will be important to design tools which limit the impact of constraint explosion.

With this in mind we showed how the unfolding technique can be made to work in the context of infinite state systems [AIN00] by adapting an algorithm described in [AČJYK96] for backward reachability analysis which can be used to verify general classes of safety properties. More precisely, we presented an unfolding algorithm for symbolic verification of unbounded Petri nets. Where previous approaches [McM95, ERV96, ER99, LB99] worked on individual markings of the net, we instead let our unfolding algorithm operate on constraints representing (potentially infinite) upward closed sets of markings. We start from a constraint describing a set of "final" markings, usually representing configurations that are undesirable during the execution of the net. From the set of final markings we unroll the net backwards, generating a *Reverse Occurrence Net (RON)*. The algorithm computes a finite postfix of the RON, which gives a complete characterization of the set of markings from which a final marking is coverable.

Given that net unfoldings represent the state space in a distributed, implicit manner the verification process is a two step process: generation of the unfolding and reasoning about the unfolding. This contrasts with traditional approaches where the verification problem is done in a single step. In his seminal work McMillan [McM93] showed that deadlock detection on complete prefix of a 1-safe petri net is NP-complete. Since the deadlock problem on petri nets is PSPACE-hard it is generally conjectured that the two step process will yield savings (in time and space) provided the unfoldings are small.

We now show how to reason with unfoldings of unbounded Petri nets by reducing the problem of deciding whether a final marking is coverable from some initial marking to satisfiability of a propositional formula. Based on the postfix algorithm, we have implemented a prototype, which we have used together with PROVER, a satisfiability checker based on the Stålmark Method [SS98], to verify safety properties for a number of examples. The main contribution of this work is an end-to-end comparison of the time and space required for the coverability problem. The comparison pits on one hand a combination of the unfolding construction and the use of PROVER for reasoning and on the other a backward reachability algorithm, which considers all interleavings. Our main conclusion is that the space and time required for reasoning based on unfoldings is significantly lesser than the space and time required for reasoning based on consideration of all interleavings.

Outline In the next section we give some preliminaries on Petri nets. In Section 3 we introduce Reverse Occurrence Nets (RONs) and unfoldings. In Section 4 we describe how the coverability problem can be reduced to a satisfiability problem which can be solved using the SAT-solver PROVER which is described in Section 5. In Section 6 we describe the implementations used in our experimentation and in Section 7 the results are reported. Finally, in Section 8 we give some conclusions and directions for future research.

2 Preliminaries

Let $\mathbb{N}$ be the set of natural of numbers. For $a, b \in \mathbb{N}$, we define $a \ominus b$ to be equal to $a - b$ if $a \geq b$, and equal to 0 otherwise. A *bag* over a set A is a mapping from A to $\mathbb{N}$. Relations and operations on bags such as $\leq$, $+$, $-$, $\ominus$, etc, are defined as usual. We use $|S|$ to denote the size of the set S.

A *net* is a triple (P, T, F) where P is a finite set of *places*, T is a finite set of *transitions*, and $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*. By a *node* we mean a place or a transition. The preset ${}^{\bullet}x$ of a node x is the set $\{y \mid (y, x) \in F\}$. The postset $x^{\bullet}$ is similarly defined. A *marking* M is a bag over P. We say that a transition t is *enabled* in a marking M if ${}^{\bullet}t \leq M$. We define a transition relation, $\longrightarrow$, on the set of markings where $M_1 \longrightarrow M_2$ if there is a $t \in T$ which is enabled in M_1 and $M_2 = M_1 - {}^{\bullet}t + t^{\bullet}$. We let $\xrightarrow{*}$ denote the reflexive transitive closure of $\longrightarrow$. We say that a marking M_2 is *coverable* from a marking M_1 if

$M_1 \xrightarrow{*} M_2'$, for some $M_2' \geq M_2$. A *net system* is a tuple $N = (P, T, F, M_{init}, M_{fin})$, where (P, T, F) is a net and M_{init}, M_{fin} are markings, called the *initial* and the *final* marking of N respectively. In this paper, we consider the *coverability problem* defined as follows.

Instance A net system $(P, T, F, M_{init}, M_{fin})$.

Question Is M_{fin} coverable from M_{init}?

Using standard methods [VW86, GW93], we can reduce the problem of checking safety properties for Petri nets to the coverability problem.

To solve the coverability problem, we perform a backward reachability analysis. We define a *backward* transition relation [AČJYK96], such that, for markings M_1 and M_2 and a transition t, we have $M_2 \rightsquigarrow_t M_1$ if $M_1 = (M_2 \ominus t^\bullet) + {}^\bullet t$. Observe that, for each marking M_2 and transition t, there is a marking M_1 with $M_2 \rightsquigarrow_t M_1$, i.e., transitions are always enabled with respect to $\rightsquigarrow$. The following result justifies the use of backward reachability:

Lemma 2.1 [AČJYK96]

1. If $M_1 \longrightarrow M_2$ and $M_2' \leq M_2$ then there is $M_1' \leq M_1$ such that $M_2' \rightsquigarrow M_1'$.
2. If $M_2 \rightsquigarrow M_1$ and $M_1' \geq M_1$ then there is M_2' such that $M_2' \geq M_2$ and $M_1' \longrightarrow M_2'$.

3 Reverse Occurrence Nets and Unfoldings

A *Reverse Occurrence Net (RON)* corresponds to a backwards "unrolling" of a net. Formally, a *RON R* is a net $(\mathsf{C}, \mathsf{E}, \mathsf{F})$ satisfying the following conditions

(i) $|c^\bullet| \leq 1$ for each $c \in \mathsf{C}$.

(ii) there is no infinite sequence of the form $c_1 \mathsf{F} e_1 \mathsf{F} c_2 \mathsf{F} \cdots$. This condition implies that there are no cycles in the RON, and that there is a set $\max(\mathsf{F})$ of nodes which are maximal with respect to F.

(iii) $\max(\mathsf{F}) \subseteq \mathsf{C}$.

In a RON, the places and transitions are usually called *conditions* and *events* respectively. A set of events $E \subseteq \mathsf{E}$ is considered to be a *configuration* if for every event e', if there is an event $e \in E$ with $e \mathsf{F}^* e'$ then $e' \in \mathsf{E}$. Intuitively, a configuration captures a set of events that could have been fired.

Consider a net system $N = (P, T, F, M_{init}, M_{fin})$ and a RON $(\mathsf{C}, \mathsf{E}, \mathsf{F})$, and let $\mu : \mathsf{C} \cup \mathsf{E} \to P \cup T$ such that $\mu(c) \in P$ if $c \in \mathsf{C}$ and $\mu(e) \in T$ if $e \in \mathsf{E}$. For $C \subseteq \mathsf{C}$, we define $\#C$ to be a marking such that, for each place p, the value of $\#C(p)$ is equal to the size of the set $\{c \in C \mid \mu(c) = p\}$. In other words $\#C(p)$ is the number of conditions in C labeled with p. We say that $(\mathsf{C}, \mathsf{E}, \mathsf{F}, \mu)$ is a *(backward) unfolding* of N if the following two conditions are satisfied:

(i) $\#\max(\mathsf{F}) = M_{fin}$, i.e., the set of conditions which are maximal with respect to F correspond to the final marking; and

(ii) μ preserves F, viz., if $(x, y) \in \mathsf{F}$ then $(\mu(x), \mu(y)) \in F$.

For a configuration E, we define $Cut(E)$ to be the set $(\{{}^\bullet e \mid e \in E\} \cup \max(\mathsf{F})) - \{e^\bullet \mid e \in E\}$. We define the marking $mark(E) = \#(Cut(E))$.

In Figure 1, we show a net system N with seven places, $p_1, \cdots, p_7$, and four transitions, $t_1, \cdots, t_4$. We also show an unfolding[1] U of N, assuming a final marking (p_1, p_7). Examples of configurations in U are $E_1 = \{e_2, e_4\}$ with $mark(E_1) = (p_1, p_2, p_3)$, and $E_2 = \{e_1, e_2, e_3, e_4\}$ with $mark(E_2) = (p_1, p_2, p_2, p_3)$.

[1] To increase readability, we show both names and labels of events in the figure, while we omit names of conditions.

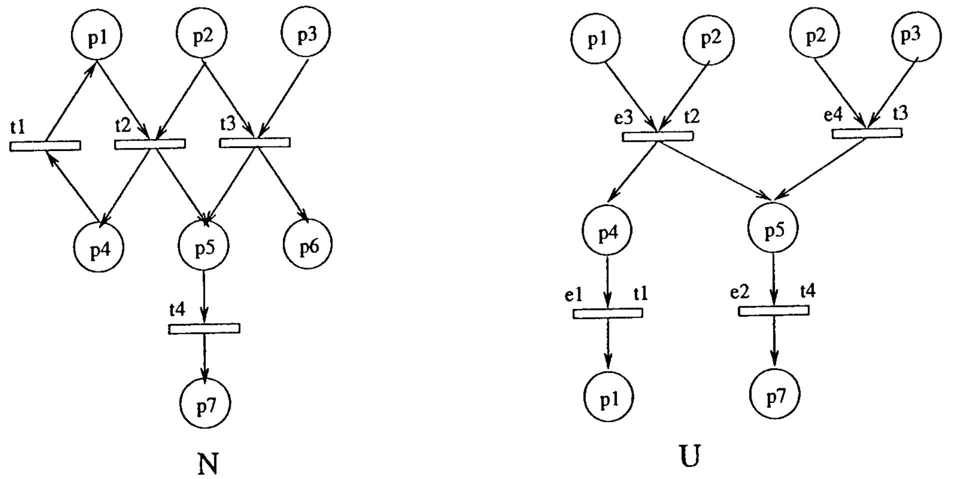


Figure 1: A net system and one of its unfoldings

Remark In [McM95, ERV96], configurations are required to be *conflict free*, i.e., for all events $e_1, e_2 \in E$ we have ${}^\bullet e_1 \cap {}^\bullet e_2 = \emptyset$. Notice that this property is always satisfied by our configurations, since we demand that $|c^\bullet| \leq 1$ for each condition.

3.1 Postfix of an unfolding

Our definition of unfolding does not preclude the fact that it could be an infinite (backward) unrolling. In practice, we wish to construct a finite postfix of the unfolding and reason with it. The construction of the finite postfix critically depends upon certain events that are called *cut-off* events. Intuitively, by backward firing cut-off events we would only be generating states that are already represented in some other part of the net.

For every event e we define its downward closure $e{\downarrow}$ to be $\{e' \mid e \; \mathsf{F}^* \; e'\}$. Clearly $e{\downarrow}$ is a configuration and has a marking associated with it. We say that a configuration E_1 covers a configuration E_2, written formally as $E_1 \prec E_2$, iff $|E_1| < |E_2|$ and $\mathit{mark}(E_1) \leq \mathit{mark}(E_2)$. From Lemma 2.1 it is easy to see that markings backward reachable from E_2 are also reachable from E_1. Consequently, it does not make any sense to explore states reachable from E_2. The importance of the size restriction comes from the fact that there could be multiplicity of tokens in a place (concurrently) and one can not be ignored for the sake of others.

An event e is a cut-off event in U if there is a configuration E in U such that $E \prec e{\downarrow}$.

In Figure 2, an algorithm which generates an unfolding $U = (\mathsf{C}, \mathsf{E}, \mathsf{F}, \mu)$ of a given net system $N = (P, T, F, M_{init}, M_{fin})$ in an incremental way is presented. In a manner similar to [ERV96], the unfolding is represented as a list of objects corresponding to conditions and events in the underlying RON. An event e is represented as an object (C, t) where t is the label $\mu(e)$ of e and C is its set $e^\bullet$ of post-conditions. A condition c is represented by an object (e, p) where p is the label $\mu(c)$ of c, and e is its (single) post-event $c^\bullet$. We observe that the flow relation F and the labeling function μ are included in the encoding.

Consider a set of conditions C of U to be t-enabled provided there exists a configuration E such that $C \subseteq \mathit{Cut}(E)$ and $0 < \#C \leq t^\bullet$, i.e., there is a configuration E such that $C \subseteq \mathit{Cut}(E)$ and all the conditions in C are in the postset of t. We will write $\mathsf{E}_t(C)$ to denote that C is t-enabled. We define $\mathit{Xtnd}(U)$ to be the set of events by which U can be extended, formally defined as follows:

$$\mathit{Xtnd}(U) = \{(C, t) \mid \mathsf{E}_t(C) \text{ and } (C, t) \notin U\}$$

Observe that the definition implies that there are no redundancies in the unfolding. In other words we will

```
Input: net system $N = (P, T, F, M_{init}, M_{fin})$, where $M_{fin} = (p_1, \cdots, p_n)$.
var $U_N$: unfolding of $N$; $X$: set of events.
begin
    $U_N := (p_1, \emptyset), \cdots, (p_m, \emptyset)$
    $X := Xtnd(U_N)$
    while $(X \neq \emptyset)$ do
        Pick and delete $e = (C, t)$ from $X$
        Add $e$ to $U$
        If $\neg cut\text{-}off(e)$ then $\forall p \in {}^\bullet t$ add a new condition $(p, e)$ to $U_N$
        $X := X \cup Xtnd(U_N)$
end
```

Figure 2: Unfolding Algorithm

not have two different events both having the same label and the same postcondition.

If the algorithm, given above, does not terminate then there would be an infinite sequence of events $\{e_i\}_{i \geq 1}$ such that for every pair of distinct events e_i and e_j we would have $mark(e_i{\downarrow})$ and $mark(e_j{\downarrow})$ are incomparable, which is not possible according to Dickson's Lemma [Dic13]. Consequently, we are assured that the unfolding construction does terminate leaving behind a structure to reason with. Note that M_{fin} is coverable from a marking M_{init} if and only if there is a configuration E in U such that $mark(E) \leq M_{init}$.

4 Checking Coverability using SAT-solvers

The problem of checking coverability from a marking M once an unfolding U has been generated is, as stated in Section 3, the problem of finding a configuration E such that $mark(E) \leq M$. Since the number of configurations of U could be very large ($|\mathcal{P}(\mathsf{E})|$ in the worst case) a brute force method based on computing the set of all configurations is not practical. We, consequently, propose to use a SAT-solver, in our case Prover, to carry out the task. This depends upon encoding the coverability problem as one of checking satisfiability of a propositional formula.

By the definition of configurations we know that for each configuration E the following holds:

(i) if an event $e \in E$ then for all events e' s.t. $e\mathsf{F}c\mathsf{F}e'$ we have $e' \in E$.

(ii) a condition $c \in Cut(E)$ if and only if $c^\bullet \subseteq E$ and for all events $e \in {}^\bullet c$, $e \notin E$.

Finally, a configuration E satisfies $mark(E) \leq M$, for a marking M, provided for each place p we have

$$\#Cut(E)(p) \leq M(p)$$

Given a net system $N = (P, T, F, M_{init}, M_{fin})$ and an unfolding $U = (\mathsf{C}, \mathsf{E}, \mathsf{F}, \mu)$ of N we can construct a formula $\mathcal{F}$, where each node x in U is represented by a variable v_x, according to the following

1. For each event $e \in \mathsf{E}$, add a conjunct $v_e \Rightarrow v_{e_1} \wedge \cdots \wedge v_{e_n}$ where $\{e_1, \cdots, e_n\}$ is the set $\{e_i \mid \exists c \in \mathsf{C} : e\mathsf{F}c\mathsf{F}e_i\}$

2. For each condition $c \in \mathsf{C}$, add a conjunct $v_c \Leftrightarrow v_{e_p} \wedge \neg v_{e_1} \wedge \cdots \wedge \neg v_{e_n}$ where $\{e_p\} = c^\bullet$ and $\{e_1, \cdots, e_n\}$ is the set $\{e_i \mid e_i\mathsf{F}c\}$

3. For each place $p \in P$, add a conjunct $LTEk(k, [v_{c_1}, \cdots, v_{c_n}])$ where $k = M_{init}(p)$, and $\{c_1, \cdots, c_n\}$ is the set $\{c_i \mid \mu(c_i) = p\}$. Furthermore, the predicate $LTEk(k, [v_1, \cdots, v_n])$ is true exactly when the number of propositions in the set $\{v_1, \cdots, v_n\}$ that are assigned true is lesser than or equal to k.

Now a model of $\mathcal{F}$ is an assignment of variables corresponding to a configuration E in the following way

- For each event e, $e \in E$ iff v_e
- For each condition c, $c \in Cut(E)$ iff v_c
- $mark(E) \leq M_{init}$

The problem of checking coverability has now been reduced to the satisfiability of the propositional formula $\mathcal{F}$, i.e., $\mathcal{F}$ is satisfiable if and only if M_{fin} is coverable from M_{init}.

The formula $\mathcal{F}$ is, of course, a simple propositional formula except for the third set of conjuncts, those containing $LTEk$. These formulas can be encoded in propositional logic; however, the naive translation would involve an exponential blow up. The SAT-solver used in this work, PROVER, has support for such predicates, which comes to our aid.

5 PROVER

To determine satisfiability we use a commercial tool, PROVER, a proof procedure for propositional logic augmented with finite domain integer arithmetic and enumerated types. The theorem prover underlying PROVER is an implementation of the Stålmarck Method [SS98] which is based on a system for natural deduction. The proof procedure has been know to be versatile and has been used with propositional formula containing as much as 350,000 connectives. PROVER deals with formulas involving all the usual logical connectives, conjunction, disjunction, implication, equivalence and negation. In addition, it also provides a number of predicates of the form LTEk($k, [f_1, \cdots, f_n]$) which are statements about how many of the n formulas $f_1, \cdots, f_n$ that are true, in this case the number of true formulas is less than or equal to k. These predicates can be defined using the basic connectives, but the naive way to do so results in a number of connectives which grows as a k-order polynomial in n, whereas the calls in PROVER use approximatively $2 * k * (n - k)$ connectives.

PROVER also deals with the arithmetic connectives, addition, subtraction, multiplication, division, remainder and negation.

6 Implementation

In our experimentation, we have compared two implementations of the unfolding algorithm and an implementation of a backwards reachability algorithm.

The implementation of the backwards reachability algorithm is a straightforward rendition of the abstract algorithm in [AČJYK96] and is as given in Figure 3. Note that this algorithm does not make use of partial-order techniques, and, therefore, considers all possible interleavings.

A technical point to note in the algorithm above is that the set Min, at the end of the k^{th} iteration, maintains the minimal elements of the set $\{M | M_{fin} \leadsto^k M\}$. Given that these minimal elements denote upward closed sets of markings, we are, again, guaranteed termination by Dickson's Lemma [Dic13]. More importantly, we wish to point out that if a marking that is smaller than M_{init} is generated then the algorithm terminates immediately without generating the entire set of minimal elements of the backward reachable set. This contrasts with our unfolding algorithm where we have to build the whole prefix before checking whether M_{init} is represented in the unfolding.

Issues in the unfolding algorithm: The implementation of the unfolding algorithm is a straight-forward rendition of the abstract algorithm given in Section 3. There were, however, two issues that need explanation;

```
Input: net system $N = (P, T, F, M_{init}, M_{fin})$.
var $Min$ : Set of minimal markings, $Q$ : Queue of markings to be considered
begin
    $Min := \emptyset$;
    $Q := \{M_{fin}\}$;
    while $(Q \neq \emptyset)$ do
        Pick and delete a marking $M$ from $Q$
        If $M \leq M_{init}$ then return "yes"
        If $\exists M' \in Min$ such that $M' \leq M$ then continue;
        Add $M$ to $Min$ while removing any $M' \in Min$ such that $M \leq M'$
        Add to $Q$ all $M'$ such that $M \rightsquigarrow M'$.
    end
    Return "No";
end
```

Figure 3: Backward Reachability Algorithm

computation of *Xtnd* and checking of termination. In the computation of *Xtnd* we maintain a queue of sets of conditions, where each set denotes a set of conditions that could hold concurrently and can, consequently, be the postset of an event. As new conditions are generated we check whether a new condition can be added to a set of conditions that is already under consideration. It is in this way that a seemingly combinatorial problem is converted to one of carrying out depth-first searches.

The two implementations that we will report upon, Unfolding 1 and Unfolding 2, use the same abstract program and the same strategy for calculating *Xtnd*. They, however, differ, on how the termination conditions are checked. In Unfolding 1 when a new event e is generated we compute $mark(e'\downarrow)$ and $|e'\downarrow|$ for all events e' in the current unfolding. Clearly, there is a lot of wasted time with this design choice but it does save on space. With Unfolding 2 with each event e' we maintain both $mark(e\downarrow)$ and $|e'\downarrow|$.

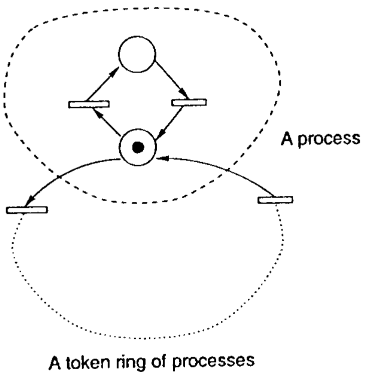
7 Experimental Results

The thesis of our experimental work is that although analysis with unfoldings is is done in two steps, constructing the unfolding and reasoning about it, the time and space required is less than for an interleaving based backward analysis for the coverability problem of unbounded Petri nets.

We will now show three examples which, in spite of being small[2], illustrates convincingly that unfoldings provide significant savings in space and time. In each case, we give the size of the Petri net in terms of the number of places, $|P|$, and the number of transitions, $|T|$. We compute (a) the maximum number of markings that need to be maintained for the traditional backward analysis [AJ98] and (b) the total number of nodes generated by the unfolding algorithm. Given that the storage requirements of a node is bounded by the storage required for a marking, comparing the number of markings from backwards analysis against the total number of nodes in an unfolding is appropriate. Furthermore, we also report on the time taken for using both the prover and the generation of the unfolding.

Example 1: The first example we consider, presented in Figure 4, is that of a simple token ring consisting of a number of processes. As we increase the number of processes the number of places and transitions increase too. Note, however, that the M_{init} and M_{fin} where so chosen that they have the same total number of tokens in each case, and, furthermore, the M_{fin} is not coverable from M_{init}. Consequently, the backwards analysis algorithm would have to compute the basis for the entire backward reachability set.

[2] Unfortunately, no large examples of unbounded petri nets are available. We checked the Petri net list's repository without much luck. Any suggestions are welcome.



# processes	$\|P\|$	$\|T\|$	Unfolding 1		Unfolding 2		Backward	
			Time $\times 10^{-2}$ sec	Space	Time $\times 10^{-2}$ sec	Space	Time $\times 10^{-2}$ sec	Space
2	4	6	3	14	1	20	1	37
4	8	12	14	30	7	44	10	147
8	16	24	79	62	26	92	284	550
16	32	48	596	126	134	188	12124	2006

Figure 4: A Simple Token Ring Network

Example 2: This is a slightly more complicated version of Example 1 designed with the aim of introducing more branching (see Figure 5). In this case, though the number of processes was changed (and thus the size of the petri net changed) the M_{fin} was kept the same. However, the initial marking was changed with every instance. Finally, in all of these cases too M_{fin} was not coverable from the M_{init}.

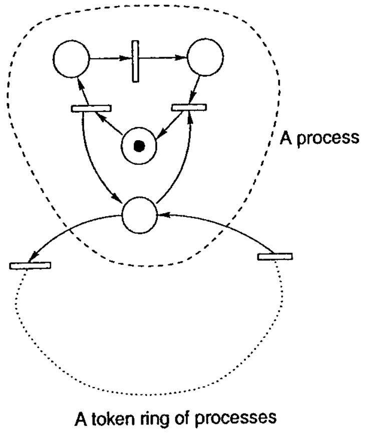
Example 3: In this example, of a buffer process, from Figure 6, the original Petri net was kept the same while the number of tokens in M_{fin} was changed with each instance while the M_{init} was kept the same. Furthermore, in all of the instances M_{fin} is coverable from M_{init}.

There are several conclusions that can be drawn from these experiments:

- Unfolding 2 is a better algorithm than Unfolding 1. While it uses approximately 1.5 times the memory that Unfolding 1 uses, the savings in time is indeed significant. The time falls by much more than a factor of 1.5.
- Both Unfolding 1 and Unfolding 2 are better than a traditional backward analysis, which uses interleaving semantics.
- The cost of using PROVER to reason about the unfolding is insignificant compared to the time, and space, needed to compute the unfolding.
- Unfoldings do offer great savings in time and space.

8 Conclusions and Future Work

We have shown how safety properties can be verified for infinite state systems modeled by unbounded Petri nets by using a combination of the unfolding technique presented in [AIN00] and a SAT-solver,



# tokens & processes	|P|	|T|	Unfolding 1		Unfolding 2		Backward	
			Time $\times 10^{-2}$ sec	Space	Time $\times 10^{-2}$ sec	Space	Time $\times 10^{-2}$ sec	Space
2	8	8	18	30	7	42	20	197
4	16	16	56	54	18	78	3053	1700
8	32	32	313	102	70	150	553324	14637
16	64	64	2611	198	409	294	*	*

Note: * implies non-termination after a reasonable amount of time.

Figure 5: A more complicated token ring network

PROVER [SS98]. We have compared this two-step process with an implementation of a single step verification that does not use unfoldings and found that it is more efficient both concerning time and space.

Since the cost of using PROVER to reason about unfoldings is insignificant compared to the cost of creating the unfolding an important direction of future research is the design of efficient data structures for implementation of the unfolding algorithm. It is also important to study the performance of the algorithm on more advanced examples.

References

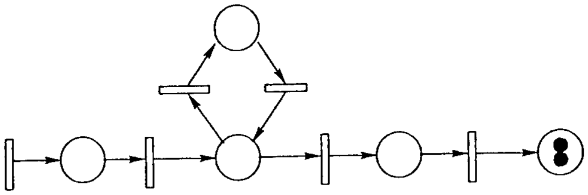
[AČJYK96] Parosh Aziz Abdulla, Karlis Čerāns, Bengt Jonsson, and Tsay Yih-Kuen. General decidability theorems for infinite-state systems. In *Proc. 11th IEEE Int. Symp. on Logic in Computer Science*, pages 313–321, 1996.

[AIN00] Parosh Aziz Abdulla, Purushothaman Iyer, and Aletta Nylén. Unfoldings of unbounded petri nets. To appear in Proc. CAV'2000, 12th Int. Conf. on Computer Aided Verification, 2000.

[AJ98] Parosh Aziz Abdulla and Bengt Jonsson. Ensuring completeness of symbolic verification methods for infinite-state systems, 1998. To appear in the journal of Theoretical Computer Science.

[Dic13] L. E. Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *Amer. J. Math.*, 35:413–422, 1913.

[ER99] J. Esparza and S. Römer. An unfolding algorithm for synchronous products of transition systems. In *Proc. CONCUR '99, 9th Int. Conf. on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 2–20. Springer Verlag, 1999.



# tokens	Unfolding 1		Unfolding 2		Backward	
	Time $\times 10^{-2}$ sec	Space	Time $\times 10^{-2}$ sec	Space	Time $\times 10^{-2}$ sec	Space
2	4	20	3	30	2	36
4	18	40	9	60	14	130
8	105	80	26	120	526	723
16	708	160	95	240	78733	6068

Figure 6: A buffer process

[ERV96] J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. In *Proc. TACAS '96, 2th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 87–106. Springer Verlag, 1996.

[GW93] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, 1993.

[LB99] R. Langerak and E. Brinksma. A complete finite prefix for process algebra. In *Proc. 11th Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 184–195. Springer Verlag, 1999.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[McM95] K.L. McMillan. A technique of a state space search based on unfolding. *Formal Methods in System Design*, 6(1):45–65, 1995.

[Pel93] D. Peled. All from one, one for all, on model-checking using representatives. In *Proc. 5th Int. Conf. on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1993.

[SS98] M. Sheeran and G. Stålmarck. A tutorial on stålmarck's proof procedure for propositional logic. In *Formal Methods in Computer-Aided Design*, volume 1522 of *Lecture Notes in Computer Science*, pages 82–99. Springer Verlag, 1998.

[Val90] A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer-Verlag, 1990.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st IEEE Int. Symp. on Logic in Computer Science*, pages 332–344, June 1986.

Evolution by Contract

Luís Filipe A. Andrade
Oblog Software SA
Alameda António Sérgio 7 – 1 A
2795 Linda-a-Velha, Portugal
landrade@oblog.pt

José Luiz L. Fiadeiro
LabMAC & Department of Informatics
Faculty of Sciences, University of Lisbon
Campo Grande, 1700 Lisboa, Portugal
jose@fiadeiro.org

Abstract. The volatility of business requirements is putting an increasing emphasis on the ability for systems to accommodate the changes required by new or different organisational needs with a minimum impact on the implemented services. We propose a discipline for software development centred around the separation between what in systems are the basic service-providers (objects) and the mechanisms (contracts) through which the behaviour of these objects is coordinated to fulfil business requirements. We show how this separation can be supported in platforms for component-based development, making it possible for systems to evolve by adding, removing or replacing contracts without having to change the rest of the system.

1 Introduction

The problem addressed and the solutions proposed in this paper result from our experience in the last five years in conceiving, developing and applying methods and tools for modelling banking applications. In banking, like in many other business activities, the pace of market evolution and the volatility of requirements have a very deep influence on organisations and their information systems. More and more, large organisations face two important problems in this respect:

- How to conceive and develop information systems in order to support the continuous evolution of the core business and the evolution of system technology?
- How to make development and evolution scalable in the context of highly volatile business domains?

For better or for worse, such organisations seek answers to these problems in the context of object-oriented development techniques, of which the UML [7] is now, *de facto*, a standard. In spite of poor support from formal methods, one has to recognise that the construction of software has become more scalable and controllable thanks to mechanisms like encapsulation, clientship and inheritance. Even if the promised land of software component markets is not exactly around the corner, increased levels of reusability can be recognised in today's development practices.

However, our experience has shown that the benefits that object-oriented techniques have brought to software *construction* cannot be extended directly to software *evolution*. Even if object-oriented techniques make it easier to build systems by putting together components in a way that reflects interactions that take place in the application domain, changes on the implemented systems that result from the need to accommodate new business rules cannot be performed in such a modular way. This is because interactions are too often "hard-wired" in the code that implements the participating objects, making it difficult to change or introduce new interactions without having to change the implementation of the objects as well. Even worse, because such changes may result in new interfaces for the participating objects, a cascade of changes throughout the implementation of the system may well be triggered to account for the other interactions in which the objects participate.

As a consequence, the evolution of an object-oriented system, understood in terms of the need to perform changes on the system after it is released, cannot be supported in a compositional way. In other words, to some extent, object-oriented development is still producing *legacy* systems as far as evolution is concerned. Yet, time-to-market and other business constraints require that information systems be able to accommodate new practices and rules with minimal impact on the core services that are already implemented, thus prompting the need for modelling techniques that enable evolution to be directly compositional over the architecture of the information system.

Our purpose in this paper is to contribute to the solution of this problem by means of a modelling primitive – that we call *contract* – and a design pattern that enables contract-based models to be implemented, in a compositional way, over component-based development frameworks like CORBA, EJB and COM.

The rationale for contracts is the realisation that, in highly volatile business domains like banking, one can distinguish between two different kinds of "entities" as far as evolution is concerned. On the one hand, we have classes of objects like *account, client,* etc, that correspond to core business entities that are relatively stable in the sense that the organisation would normally prefer not to have to touch them, often because they really constitute *legacy* assets that are important to preserve. On the other hand, we have all the business products like account packages (different types of savings accounts, credits, etc) that keep changing because they determine the competitive edge of the organisation. These products require a layer of coordination to be established over the functionalities of the business entities so that the overall behaviour desired for the system can emerge.

When systems are conceived as collections of interacting objects, the problems that we have just identified require that we be able to express, and make available as first-class citizens, the constraints and the rules that capture the business requirements of the application domain. Because business rules determine the way object behaviour and interaction needs to be coordinated, it is necessary that these coordination aspects be available explicitly in the system models so that they can be changed, as a result of modifications that occur at the level of the business requirements, without having to modify the basic objects that compose the system. The purpose of contracts, as used in this paper, is to provide mechanisms for that layer of coordination to be modelled and implemented in a compositional way. In [4], we have emphasised the static modelling aspects of the concept and shown how contracts can be introduced as an extension of the UML [7]. In this paper, we focus on the dynamic aspects and present contracts as a means of structuring the evolution of systems.

2 An example

We will use an example from banking in order to motivate the notion of contract that we are proposing. The notation that we use in the examples is a shortened version of the Oblog language [http://www.oblog.com] that we have been developing for object-oriented modelling. An example of a class specification is given below for bank accounts.

```
class Account
operations
        class
            Create(client:Customer,iAmount:Integer)
        object
            Deposit(amount:Integer)
            Withdrawal(amount:Integer)
            Balance() : Integer;
            Transfer(amount:Integer, target:Account);
body
        attributes
            number : Integer;
            balance : Integer := 0
        methods
            Deposit is set Balance := Balance+amount
            Withdrawal is set Balance := Balance-amount
            Transfer is {    call target.Deposit(amount);
                             call self.Withdrawal(amount) }
end class
```

In Oblog, a class specification includes a section in which the interface operations are declared. We distinguish between class and object operations: the former are used for managing the population of the class as a whole and the latter apply to each specific instance. Each operation is declared with a list of input and output parameters and a specification of its required behaviour in terms of pre/post conditions (omitted in the example for simplicity). In the case of the bank account, the operations that were chosen are self-explanatory.

The body section of a class specification identifies the attributes that define the state of the instances as well as the implementations of the operations (called methods). Methods can be guarded with state-conditions like in Dijkstra's guarded commands. In fact, one may wonder why a guard has not been specified for `withdrawal` restricting this method to occur in states in which the amount to be withdrawn can be covered by the balance. The answer to this question is a good motivation for contracts.

Assigning the guard Balance≥amount to withdrawal can be seen as part of the specification of a business requirement and not necessarily of the functionality of a basic business entity like account. Indeed, the circumstances under which a withdrawal will be accepted can change from customer to customer and, even for the same customer, from one account to another depending on its type.

One could argue that, through inheritance, this guard could be changed in order to model these different situations. However, there are two main problems with the use of inheritance for this purpose. On the one hand, it views objects as white boxes in the sense that adaptations like changes to guards are performed on the internal structure of the object. From the point of view of evolution, this is not desirable. On the other hand, from the business point of view, the adaptations that make sense may be required on classes other than the ones in which the restrictions were implemented. In the example above, this is the case when it is the type of client, and not the type of account, that determines the nature of the guard that applies to withdrawals.

Hence, it makes more sense for business requirements of this sort to be modelled explicitly outside the classes that model the basic business entities. Our proposal is that guards like the one discussed above should be modelled as *contracts* that can be established between clients and accounts. In fact, we will provide mechanisms for such contracts to be *superposed* on existing implementations of clients and accounts, considered as black boxes, so that contracts can be added and deleted in a flexible way *(plug and play)*, reflecting the evolution of the business domain.

The example given above motivates the advantage of modelling, as first-class entities, the mechanisms that control the usage of given objects (contracts as controllers). The example below aims at illustrating cases in which a layer of coordination among different objects is required that is active in its own right.

One of the latest products to appear in the banking area can be called "the flexible package". This is a mechanism via which automatic transfers are made between a checking account and a savings account of the same client: from savings to checking when the balance goes below a certain threshold, and from checking to savings when the balance goes above a certain threshold.

Like before, the application of traditional object-oriented techniques for adding this new feature to the system would probably raise a number of problems. The first one concerns the decision on where to place the code that is going to perform the transfers: the probable choice would be the checking account because that is where the balance is kept. Hence, the implementation of account would have to be changed. The "natural" solution would be to assign the code to a new association class between the two accounts but, again, current techniques for implementing association classes require the implementations of the participating classes to be changed because the associations are implemented via attributes.

Another problem is concerned with the handling of the synchronisation of the transfers. If the transfers are not coded in the methods of the accounts, there is no way in which the whole process can be dealt with atomically as a single transaction. Again, what is required is a mechanism via which we can superpose a layer of coordination that is separate from the computations that are performed locally in the objects. This is exactly the purpose of contracts as detailed in the next section.

3 Contracts

From a static point of view, a contract defines an *association class* in the sense of the UML (i.e. an association that has all the attributes of a class) but the way interaction is established between the partners is more powerful: it provides a *coordination role* that is closer to what is available for configurable distributed systems and software architectures in general.

Another useful analogy is with architectural connectors [2]. A contract consists, essentially, of a collection of role classes (the partners in the contract) and the prescription of the coordination effects (the glue in the terminology of software architectures) that will be superposed on the partners.

In Oblog, contracts are defined as follows:

```
contract <name>
       partners <list-of-partners>
       invariant <the relation between the partners>
       constants
       attributes
       operations
       coordination <interactions-with-partners>   behaviour < behaviour being superposed>
end contract
```

The instances of the partners that can actually become coordinated by instances of the contract are determined through a set of conditions specified as invariants. The typical case is for instances to be required to belong to some association between the partners.

Each interaction under "coordination" is of the form

```
<name> :   when <condition>
           do <set of actions>
           with <condition>
```

The name of the interaction is necessary for establishing an overall coordination among the various interactions and the contract's own actions. This is similar to what happens in parallel program design languages like Interacting Processes [14]. The condition under "when" establishes the trigger of the interaction. Typical triggers are the occurrence of actions in the partners. The "do" clause identifies the reactions to be performed, usually in terms of actions of the partners and some of the contract's own actions. Together with the trigger, the reactions of the partners constitute what we call the synchronisation set associated with the interaction. Finally, the "with" clause puts further constraints on the actions involved in the interaction, typically further preconditions.

The intuitive semantics (to be further discussed in the following sections) is that, through the "when" clause, the contract intercepts calls to the partners or detects events in the partners to which it has to react. It then checks the "with" clause to determine whether the interaction can proceed and, if so, coordinates the execution of the synchronisation set. All this is done atomically.

An example can be given through the account packages already discussed. The traditional package, by which withdrawals require that the balance be greater than the amount being withdrawn, can be specified as follows:

```
contract Traditional package
        partners x : Account; y : Customer;
        invariants ?owns(x,y)=TRUE;
        coordination
             tp: when y.calls(x.withdrawal(z))
                 do x.withdrawal(z)
                 with x.Balance() > z;
end contract
```

Notice that, as specified by the invariant, this contract is based on an ownership association that must have been previously defined. This contract involves only one interaction. It relates calls placed by the customer for withdrawals with the actual withdrawal operation of the corresponding account. The customer is the trigger of the interaction: the interaction requires every call of the customer to synchronise with the withdrawal operation of the account but enables other withdrawals to occur outside the interactions, e.g. by other joint owners of the same account. The constraint is the additional guard already discussed. Notice that the constraint applies only to the identified pair of customer and account, meaning that other owners of the same account may subscribe to different contracts.

The notation involving the interaction in this example is somewhat redundant because the fact that the trigger is a call from the customer to an operation of the account immediately identifies the reaction to be performed. In situations like this, Oblog allows for abbreviated syntactical forms of interaction. However, in the paper, we will consistently present the full syntax to make explicit the various aspects involved in an interaction. In particular, the full syntax makes it explicit that the call put by the client is intercepted by the contract, and the reaction, which includes the call to the supplier, is coordinated by the contract. Again, we stress that such interactions are atomic, implying that the client will not know what kind of coordination is being superposed. From his point of view, it is the supplier that is being called.

As already explained, the purpose of contracts is to externalise the interactions between objects, making them explicit in the conceptual models, thus reflecting the business rules that apply in the current state. Hence, contracts may change as the business rules change, making system evolution compositional with respect to the evolution of the application domain. For instance, new account packages may be introduced that relax the conditions under which accounts may be overdrawn:

```
contract VIP package
        partners x : Account; y : Customer;
        constants CONST_VIP_BALANCE: Integer;
        attributes Credit : Integer;
        invariants
             ?owns(x,y)=TRUE;
             x.AverageBalance() >= CONST_VIP_BALANCE;
```

```
          coordination
              vp: when y.calls(x.withdrawal(z))
                  do   x.withdrawal(z)
                  with x.Balance() + Credit() > z;
   end contract
```

Notice that, on the one hand, we have strengthened the invariant of the contract, meaning that only a restricted subset of the population can subscribe to this new contract. On the other hand, the contract weakens the guard imposed on withdrawals, meaning that there are now more situations in which clients can withdraw money from their accounts.

In general, we allow for contracts to have features of their own. This is the case of the contract above for which an attribute and a constant were declared to account for the credit facility. It is important to stress that such features (including any additional operations) are all private to the contract: they cannot be made available for interaction with objects other than the partners. Indeed, the contract does not define a public class.

Our last example models the flexible package that we have already motivated.

```
  contract Flexible package
         partners c, s : Account;
         attributes min, max : Integer;
         invariants c.owner=s.owner;
         coordination
             putfunds:when calls(c.Deposit(z))
                      do {    if c.Balance()+z > max then {
                                      c.Deposit(z); c.Transfer(c.Balance()-max,s)} }
             getfunds:when calls(c.Withdrawal(z))
                      do {    if c.Balance()-z < min then {
                                      s.Transfer(min-c.Balance(),c); c.Withdrawal(z)}       }
  end contract
```

4 Semantical aspects

The intuitive semantics of contracts can be summarised as follows:

- Contracts are added to a system by identifying the instances of the partner classes to which they apply; these instances may belong to subclasses of the partners; for instance, in the case of the flexible package, both partners were identified as being of type account but, normally, they will be applied to two subclasses of account: a checking account and a savings account. The actual mechanism of identifying the instances that will instantiate the partners and superposing the contract is outside the scope of the paper. In Oblog, this can be achieved directly as in languages for reconfigurable distributed systems [20], or implicitly by declaring the conditions that define the set of those instances.
- Contracts are superposed on the partners taken as black-boxes: the partners in the contract are not even aware that they are being coordinated by a third party. In a client-supplier mode of interaction, instead of interacting with a mediator that then delegates execution on the supplier, the client calls directly the supplier; however, the contract "intercepts" the call and superposes whatever forms of behaviour are prescribed; this means that it is not possible to bypass the coordination being imposed through the contract because the calls are intercepted;
- The same transparency applies to all other clients of the same supplier: no changes are required on the other interactions that involve either partner in the contract. Hence, contracts may be added, modified or deleted without any need for the partners, or their clients, to be modified as a consequence;
- The interaction clauses in a contract identify points of *rendez-vous* in which actions of the partners and of the contract itself are synchronised; the resulting synchronisation set is guarded by the conjunction of the guards of the actions in the set and requires the execution of all the actions in the set;
- The effect of superposing a contract is cumulative; because the superposition of the contract consists, essentially, of synchronous interactions, different contracts will superpose their coordinating behaviour, achieving a cumulative effect. For instance, in the example of the flexible package, the transfers can also be subject to contracts that regulate the discipline of withdrawals for the particular account and client.

Contracts, as motivated in the previous sections, draw from several mechanisms that have been available for sometime in Software Engineering, which further clarifies the rationale for the semantics that we gave above:

- Contract-based development is based on the idea of separating *computation* from *coordination*, i.e. of making clear what components in a system provide the functionalities on which services are based, and what mechanisms are responsible for coordinating the activities of those components so that the desired behaviour emerges from the interactions that are established. This idea has been promoted by researchers in the area of Programming Languages who have coined the term "Coordination Languages and Models" [e.g. 16].
- The importance of having these coordination mechanisms available as first-class entities and as units of structure in system models and designs was inspired by the role played by connectors in software architectures [23]. This is why in [4] we proposed contracts as a semantic primitive enriching the notion of association class.
- The ability for objects to be treated as black-box components and, hence, for contracts to be dynamically added or removed from a system without having to recompile the partners, is achieved through the mechanism of superposition (or superimposition) developed in the area of Parallel Program Design [9,14,18].

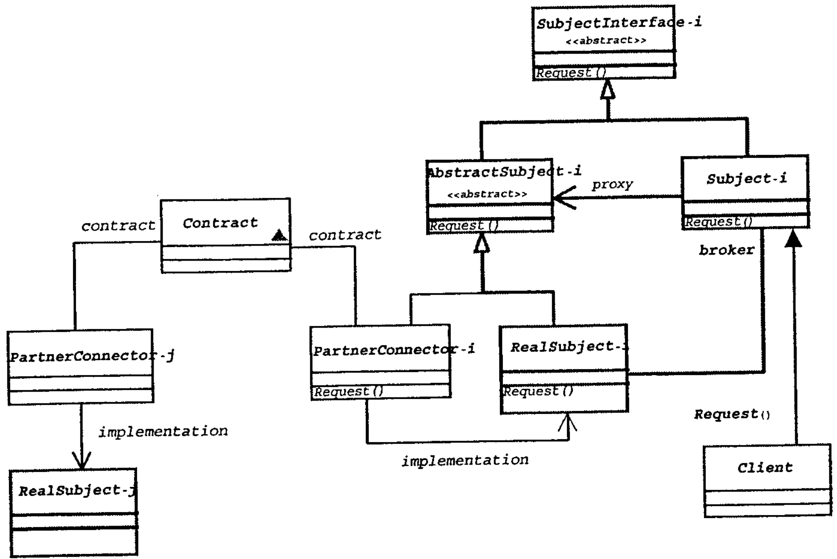
It is important to state that the contribution of superposition techniques for component adaptation in a black-box style has also been recognised in [8]. The relationship between coordination languages and software architectures has also been recognised by several authors, to the point in which joint workshops are now organised by the two communities. Our main contribution has been to integrate these three aspects into a primitive that can be used very effectively for developing and evolving systems in a way that is compositional with respect to the evolution of the business rules in the application domain.

In fact, our contribution has gone a step further in that, over several years, we have developed a mathematical semantics that brings all these aspects together: coordination, superposition, and architectures. The presentation of this semantics is outside the scope of this paper. Please consult [11] for the formalisation of different kinds of superposition that lead to the identification of different kinds of architectural connectors (regulators, monitors, etc); [12] for a formalisation of architectural connectors based on the previous formalisation of superposition, which includes the semantics of instantiation of the partners (roles); [13] for the coordination aspects as related to superposition and the application to software architectures; [25] for the application to dynamic reconfiguration, including the definition of algebraic operations on architectural connectors that are directly applicable to contracts; and [4] for the first formalisation of contracts as presented herein, and their relationship to the UML.

5 The contract design pattern

As already explained in the previous sections, a contract works as an active agent that coordinates the contract partners. In this section, we are concerned with the way these coordination mechanisms can be implemented. When defining an implementation, we need to have in mind that, as motivated in the introduction, we should be able to superpose a contract to given objects in a system and coordinate their behaviour as intended *without having to modify the way these objects are implemented.* This degree of flexibility is absolutely necessary when the implementation of these objects is not available or cannot be modified, as in legacy systems. It is also a distinguishing factor of contracts when compared with existing mechanisms for modelling object interaction, and one that makes contracts particularly suited in business domains where the ability to support the definition and dynamic application of new forms of coordination is a significant market advantage.

Different standards for component-based software development have emerged in the last few years, among which CORBA, JavaBeans and COM are the current trend in industry. However, none of these standards provide a convenient and abstract way of supporting superposition as a first-class mechanism. Because of this, we propose our solution as a design pattern. This pattern exploits some widely available properties of object-oriented programming languages such as polymorphism and subtyping, and is based on other well known design patterns, namely the Broker, and the Proxy or Surrogate [15].



The class diagram below depicts the proposed pattern. In what follows, we explain, in some detail, its basic features, starting with the participating classes.

SubjectInterface-i – as the name indicates, it is an abstract class (type) that defines the common interface of services provided by AbstractSubject-i and Subject-i.

Subject-i – This is a concrete class that implements a broker maintaining a reference that lets the subject delegate received requests to the abstract subject (AbstractSubject-i) using the polymorphic entity proxy. At runtime, this entity may point to a RealSubject-i if no contract is involved, or point to a PartnerConnector-i that links the real subject to the contracts that coordinate it.

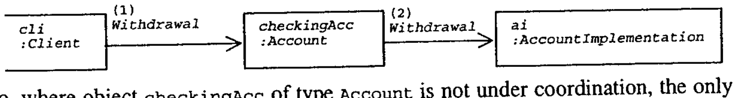
AbstractSubject-i – This is an abstract class that defines the common interface of RealSubject-i and PartnerConnector-i. The interface is inherited form SubjectInterface-i to guarantee that all these classes offer the same interface as Subject-i (the broker) with which real subject clients have to interact.

RealSubject-i – This is the concrete domain class with the business logic that defines the real object that the broker represents. The concrete implementation of provided services is in this class.

PartnerConnector-i – This class maintains the connection between contracts and the real object (RealSubject-i) involved as a partner in the contract. Adding or removing other contracts to coordinate the same real object does not require the creation of a new instance of this class but only of a new association with the new contract and an instantiation link with the existing instance of PartnerConnector-i. This means that there is only one instance of this class associated with one instance of RealSubject-i.

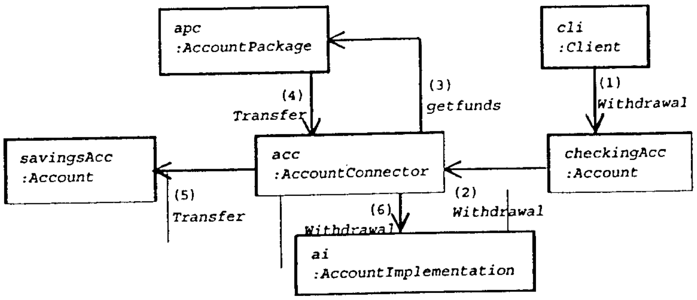
Contract – This is a coordination object that is notified and takes decisions whenever a request is invoked on a real subject or when it change its state.

If there are no contracts coordinating a real subject, the contract pattern can be simplified and only the classes that are not dimmed in the figure become necessary. The introduction of a contract implies the creation of instances for the dimmed classes and associations. The following is a possible object diagram when no coordination contract is defined, which means that there are no Contract and no PartnerConnector instances.



In this scenario, where object `checkingAcc` of type `Account` is not under coordination, the only overhead imposed by the pattern is an extra call from the broker to the real object (`AccountImplementation`). Introducing a new contract to coordinate interaction with objects of type `Account` implies only modifications on the object that plays the role of broker, i.e. in the object `checkingAcc`, making its proxy become a reference to the object that plays the role of a contract partner connector, i.e. the object `acc` of type `AccountConnector`, as seen in the following interaction diagram. Doing only this minor modification, neither the code of clients (e.g. object `cli`) nor the code of the broker `checkingAcc` and the real object `ai` need to be modified in order to accommodate the new behaviour established by adding the contract apc of type AccountPackage.

The new behaviour introduced by contract apc is described in the object interaction diagram below. This diagram shows how the contract superposes a new behaviour when requests of type Withdrawal() are invoked on an object of type Account.



In this diagram we can see that once Withdrawal() is sent to the object broker checkingAcc, it delegates its execution on the proxy reference (in this case on acc instead of checkingAcc, as seen on the previous figure). In acc the implementation of subject services has the following format

```
Request() IS
         IF   (a coordination guard holds)
         THEN Execute the coordination do code
         ELSE Execute the original code of Request
```

That is to say, before the partner connector acc gives rights to the real object implementation ai to execute the request, it intercepts the request and gives right to the contract apc to decide if the request is valid and perform other actions. This interception allows us to impose other contractual obligations on the interaction between the caller and the callee. This is the situation of the first model discussed in section 3 where new preconditions were established between Account Withdrawals and their Customers. On the other hand, it allows the contract to perform other actions before or after the real object executes the request. This is the situation of the the coordination equation named getfunds established by the contract Flexible package in section 3. Only if the contract authorises can the connector ask the object implementation ai to execute and commit, or undo execution because of violation of post-conditions established by the contract.

As stated at the beginning of this section, current component-based technology does not provide a convenient way for coordinating components. The benefit of having this form of coordination available as a primitive construction when specifying components and their interactions is that it avoids the burden of having to code such a pattern. In the meanwhile, tools which, like Oblog, provide automatic code generation from high level specifications, must hide the implementation complexity of coordination, allowing the developer just to specify the contract itself.

6 Contract-based development

The object technology market is growing very rapidly, increasing the offer of low-level tools and technologies for the fine-grain parts of information system development – *development in the small*. However, the current practice of object-oriented software development methods in what concerns the large-grain parts, namely architectures – *development in the large* – still involves the sequential use of analysis and design techniques and tools. For most large-scale systems, this is quite a wasteful process to follow because it ignores the existence of software components, design models and environment frameworks.

A new trend is emerging in software development that is based on a growing belief that analysis and design should be based on predefined frameworks of skeletal applications, components, and design patterns that can be easily customised and integrated. The only hope for organisations to be able to face the challenges of the very fast market evolution that we are already witnessing is, clearly, to adopt such a strategy. Therefore, it seems clear that the integration and coordination of the different parts of information systems, at all levels of granularity, is one of the major challenges that software development needs to face.

We believe that a good basis for meeting this challenge can be provided by adopting a development strategy based on three subprocesses:

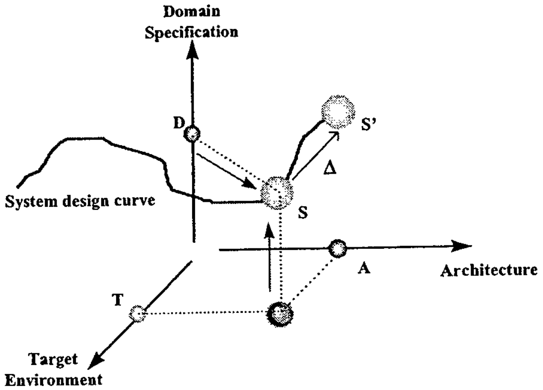
1. the construction of individual components implementing the domain objects;
2. the construction of contracts that play the roles of coordination agents responsible for implementing the configurable business rules for specific domains;
3. the construction of generic component connectors for specific architectures.

Given that contracts can be dynamically superposed on existing components, the word 'construction' above can be replaced by 'identification'. Indeed, because contracts treat the partner instances as black-boxes, this approach promotes reuse at all levels. Moreover, it facilitates the integration of third-party, closed components, namely legacy systems.

The advantages that we have identified above in terms of supporting evolution apply, as stated in the introduction, to changes that are triggered by the evolution of the business rules that apply to the system at hand. As we all know, changes to the system can be triggered by many other factors. Our experience has shown that implementing an information system involves making decisions that may be decomposed according to three different dimensions, as depicted in the figure below.

- **domain of the problem being solved** – this dimension is concerned with the description of the problem at hand, leading to an ideal model, free from any details concerning implementation. This model is defined using a specification language.
- **system architecture description** – this dimension is concerned with the way the system is structured in terms of components and interconnections. Modules are vital for dividing large specifications into parts, and to specify these parts with sufficient precision that one can construct each part knowing only the specification of the other parts, which points to a component-based approach [24]. The nature of the components that are needed, and the form in which they need to be interconnected, are influenced by infrastructural constraints like the distribution strategy or the type of interaction with the environment;
- **software chosen for the implementation** – this dimension is concerned with the technology used to implement the business problem according to the chosen architecture.

Ideally, a choice taken with respect to one of these dimensions should not interfere with the decisions related to the other. This independence enables designers to change a previous decision in one of the dimensions without having to reconsider the decisions taken in the other, thus achieving a significant degree of flexibility.



As part of future work, we intend to study the way a contract-based approach can influence development and evolution not only along the Domain Specification, but also the Architecture dimension. For that purpose, we can capitalise on work already done in using notions similar to contracts at these lower levels of development. In the next section, we identify some of these approaches.

7 Related work

Several authors have already made similar observations about the need to make explicit and available, as first-class citizens, the rules that govern the behaviour of systems, namely in [19], which became the subject matter of the ISO General Relationship Model (ISO/IEC 10165-7). The notion of contract that we proposed in the paper is in the spirit of this work but adds to it the evolutionary aspects that it inherits from the architectural approaches, and the concurrency and synchronisation aspects that it inherits from the notion of superposition as used for parallel program design.

In section 4, we have also mentioned, and gave evidence to the effect, that our approach capitalises on a number of proposals that have been made in the areas of Programming Languages (coordination languages and models), Parallel Program Design (superposition), and Software Engineering (software architectures). In fact, we view the notion of contract as a synthesis of these three aspects which, together with the experience that we have gathered in information system development, justify an investment in methodologies and tools for supporting contract-based development.

Having acknowledged the sources on which our proposal is grounded we must, nevertheless, emphasise that other approaches can be found in the literature that, in many ways, are similar in spirit or capture some of the intuitions that we had when developing contracts.

We should start by saying that the term 'contract' itself has been quite overloaded:

- there are, of course, contracts in the sense of Meyer [21]; their purpose is to support the development of object methods in the context of client-supplier relationships between objects. Therefore, they apply, essentially, to the construction of systems. In fact, in what concerns evolution, Meyer's contracts are at the origin of some of the problems that we have identified in the introduction and which made us point out that even object-oriented development techniques are producing legacy systems: by adopting clientship as the basic discipline for object interconnection, a bias is introduced in the way business rules get coded up, assigning to the supplier side the reponsibility for accommodating changes that, from the point of view of the business rules, belong to the client. This was the case of the flexible withdrawals for VIP customers: by placing the contract on the supplier side (the account), the new rules are more easily modelled as specialisations of account whereas, in the application domain, they reflect specialisations of the client. In [4] we have discussed this matter with more detail. In summary, our opinion is that Meyer's notion of contract does not scale up to system evolution.
- A notion of contract that applies to behaviours and not to individual operations or methods is the one developed in [17]. The aim of contracts as developed therein is to model collaboration and behavioural relationships between objects that are jointly required to accomplish some task. The emphasis, however, is in software construction, not so much in evolution.
- A notion of contract can also be found in [5] that emerged in the context of the action-systems approach. Like in our approach, it promotes the separation between the specification of what actors can do in a system and how they need to be coordinated so that the computations required of the system are indeed part of the global behaviour. The architectural and evolutionary dimensions are not explored as such.

Besides these related notions of contract, other approaches can be found in the literature that explore the use of architectures in evolution:

- The importance of architectures for run-time software evolution has been recently pointed out [22]. The authors highlight the role of architectural connectors in mediating and governing interactions among components, which is essential for facilitating system evolution in the sense that we motivated in the paper. However, the authors concentrate too much in presenting design and implementation solutions in the form of a tool set for a Java-C2 combination, missing the abstraction power that superposition provides as a general technique for enabling contract (or connector) based evolution, and the setting-up of semantic primitive which, like contracts, can be used in conjunction with general languages for object-oriented development like the UML.
- Another recent architectural-approach to evolution that is important to mention is the one developed by [10] in which a notion of contract is explicitly developed for managing exchanges between units of evolution. These units, called zones, encapsulate processes, objects and types at design and run-time so that the contents of one zone can be evolved without affecting the code in another zone. Contracts make the inter-relationships and communication between zones explicit by providing mechanisms – "change absorbers" – for transforming calls made on a pre-evolution type from outside the zone into a call onto the evolved type. The emphasis is, therefore, very much on "lower-level" aspects of managing change during run-time evolution which, nevertheless, is a good indication of the importance of architecture-based approaches for all levels of evolution.

We have also mentioned already that we share with [8] the belief that superposition (or superimposition) is the enabling technique for a truly "black-box", "plug-and-play" approach to evolution. A basic difference between the proposed models is that, whereas our purpose is to support connectors, i.e. mechanisms for interconnecting components, the goal in [8] is to adapt software components. It results that [8] proposes a layered model of wrappers reflecting the successive adaptations to which the component is subjected. Instead, our proposal does not rely on wrappers but on proxys and brokers that facilitate the dynamic reconfiguration of the context in which the component is being used.

To summarise, it is clear that, by bringing to bear techniques from Coordination Languages and Models, Software Architectures, and Parallel Program Design, the notion of contract-based software evolution that we proposed in this paper shares many of the advantages of similar approaches that have been proposed in the literature exploring some of these combinations (but not all of them). The experience that we have gathered in applying the concept in real-world projects suggests that contracts are not only an effective mechanism for enabling a flexible evolution of business products, but also a powerful modelling primitive that analyst find useful at the earlier stages of domain modelling, at least in the business areas that we have already mentioned.

8 Concluding remarks

In this paper, we presented a discipline for software development and evolution centred around the notion of contract. Contracts were motivated by the need that we have experienced, when developing and applying tools in business areas, to separate the domain concepts (objects) from the business rules that regulate their behaviour. This separation recognises that there are two different dynamics in system evolution: changes to the way components operate and changes to the way components are interconnected. The former requires a "white box" view of components and is supported by object-oriented techniques like inheritance. The latter requires a "black box" view of components and, so far, has been lacking adequate support. The aim of contracts, as proposed in the paper, is to support this latter view towards more flexible ways of system evolution.

Contracts bring to bear techniques developed in the area of Coordination Languages and Models, Reconfigurable Distributed Systems, Software Architectures and Parallel Program Design. More precisely, contracts promote the separation of the coordination aspects that regulate the way objects interact in a system, from the way objects behave internally; they fulfil a role similar to architectural connectors in the sense that they make available these coordination features as first-class citizens. Contracts are based on superposition mechanisms for supporting forms of dynamic reconfiguration of systems. These mechanisms enable contracts to be added or replaced without the need to change the objects to which they apply.

As a consequence, we can claim that new levels of flexibility have been added to the development process, promoting *plug and play*, and a better integration and coordination of third-party, closed components (e.g. legacy systems). This, we believe, will bring us one step closer to a real industry of software components.

References

1. G.Agha, *ACTORS: A model of Concurrent Computation in Distributed Systems*, MIT Press 1986.
2. R.Allen and D.Garlan, "A Formal Basis for Architectural Connectors", *ACM TOSEM*, 6(3), 1997, 213-249.
3. L.F.Andrade, J.Gouveia, P.Xardone and J.Camara, "Architectural Concerns in Automating Code Generation", in *Proc. TC2 First Working IFIP Conference on Software Architecture*, P. Donohoe (ed), Kluwer Academic Publishers.
4. L.F.Andrade and J.L.Fiadeiro, "Interconnecting Objects via Contracts", in *UML'99 – Beyond the Standard*, R.France and B.Rumpe (eds), LNCS 1723, Springer Verlag 1999, 566-583.
5. RJ.Back, L.Petre and I.Paltor, "Analysing UML Use Cases as Contracts", in *UML'99 – Beyond the Standard*, R.France and B.Rumpe (eds), LNCS 1723, Springer Verlag 1999, 518-533.
6. L. Bass, P.Clements and Rick Kasman, *Software Architecture in Practice*, Addison Wesley 1998
7. G.Booch, J.Rumbaugh and I.Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley 1998.
8. J.Bosch, "Superimposition: A Component Adaptation Technique", *Information and Software Technology* 1999
9. K.Chandy and J.Misra,*Parallel Program Design - A Foundation*, Addison-Wesley 1988.

10. H.Evans and P.Dickman, "Zones, Contracts and Absorbing Change: An Approach to Software Evolution", in *Proc. OOPSLA'99*, ACM Press 1999, 415-434.

11. J.L.Fiadeiro and T.Maibaum, "Categorical Semantics of Parallel Program Design", *Science of Computer Programming* 28, 1997, 111-138.

12. J.L.Fiadeiro and A.Lopes, "Semantics of Architectural Connectors", in *TAPSOFT'97*, LNCS 1214, Springer-Verlag 1997, 505-519.

13. J.L.Fiadeiro and A.Lopes, "Algebraic Semantics of Coordination, or what is in a signature?", in *AMAST'98*, A.Haeberer (ed), Springer-Verlag 1999

14. N.Francez and I.Forman, *Interacting Processes*, Addison-Wesley 1996.

15. E.Gamma, R.Helm, R.Johnson and J.Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley 1995.

16. D.Gelernter and N.Carriero, "Coordination Languages and their Significance", *Communications ACM* 35, 2, pp. 97-107, 1992.

17. R.Helm, I.Holland and D.Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems", in *Proc. OOPSLA'90/ECOOP'90*, ACM Press 1990, 169-180

18. S.Katz, "A Superimposition Control Construct for Distributed Systems", ACM TOPLAS 15(2), 1993, 337-356.

19. H.Kilov and J.Ross, *Information Modeling: an Object-oriented Approach*, Prentice-Hall 1994.

20. J.Magee and J.Kramer, "Dynamic Structure in Software Architectures", *in 4th Symp. on Foundations of Software Engineering*, ACM Press 1996, 3-14.

21. B.Meyer, "Applying Design by Contract",*IEEE Computer*, Oct.1992, 40-51.

22. P.Oreizy, N.Medvidovic and R.Taylor, "Architecture-based Runtime Software Evolution", in *Proc. ICSE'98*, IEEE Computer Science Press 1998.

23. D.Perry and A.Wolf, "Foundations for the Study of Software Architectures", *ACM SIGSOFT Software Engineering Notes* 17(4), 1992, 40-52.

24. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison Wesley 1998

25. M.Wermelinger and J.L.Fiadeiro, "Towards an Algebra of Architectural Connectors: a Case Study on Synchronisation for Mobility", in *Proc. 9th International Workshop on Software Specification and Design*, IEEE Computer Society Press 1998, 135-142.

"Lightweight" Semantics Models for Program Testing and Debugging Automation

(Extended Abstract)

Mikhail Auguston
Computer Science Department, New Mexico State University,
Las Cruces, NM 88003, USA
mikau@cs.nmsu.edu
http://www.cs.nmsu.edu/~mikau

1 Introduction

We suggest an approach to the development of software testing and debugging automation tools based on precise program behavior models. The program behavior model is defined as a set of events (event trace) with two basic binary relations over events -- precedence and inclusion, and represents the temporal relationship between actions. A language for the computations over event traces is developed that provides a basis for assertion checking, debugging queries, execution profiles, and performance measurements.

The approach is nondestructive, since assertion texts are separated from the target program source code and can be maintained independently. Assertions can capture both the dynamic properties of a particular target program and can formalize the general knowledge of typical bugs and debugging strategies. An event grammar provides a sound basis for assertion language implementation via target program automatic instrumentation. Event grammars may be designed for sequential as well as for parallel programs. The approach suggested can be adjusted to a variety of programming languages. We illustrate these ideas on examples for the Occam and C programming languages.

Dynamic program analysis is one of the least understood activities in software development. A major problem is still the inability to express the mismatch between the expected and the observed behavior of the program on the level of abstraction maintained by the user [9]. In other words, a flexible and expressive specification formalism is needed to describe properties of the software system's implementation. Program testing and debugging is still a human activity performed largely without any adequate tools and consuming more than 50% of the total program development time and effort [8]. Debugging concurrent programs is even more difficult because of parallel activities, non-determinism and time-dependent behavior.

One way to improve the situation is to partially automate the debugging process. Precise *model of program behavior* becomes the first step towards debugging automation. It appears that traditional methods of programming language semantics definition don't address this aspect. In building such a model several considerations were taken in account. The first assumption we make is that the model is discrete, i.e. comprises a finite number of well-separated elements. This assumption is typical for Computer Science methods used for static and dynamic analysis of programs. For this reason the notion of *event* as an elementary unit of action is an appropriate basis for building the whole model. The event is an abstraction for any detectable action performed during the program execution, such as a statement execution, expression evaluation, procedure call, sending and receiving a message, etc.

Actions (or events) are evolving in time and the program behavior represents the temporal relationship between actions. This implies the necessity to introduce an ordering relation for events. Semantics of parallel programming languages and even some sequential languages (such as C) don't require the total ordering of actions, so *partial event* *ordering* is the most adequate method for this purpose [11].

Actions performed during the program execution are at different levels of granularity, some of them include other actions, e.g. a subroutine call event contains statement execution events. This consideration brings to our model *inclusion relation*. Under this relationship events can be hierarchical objects and it becomes possible to consider program behavior at appropriate levels of granularity.

Finally, the program execution can be modeled as a set of events (*event trace*) with two basic relations: partial ordering and inclusion. The event trace actually is a model of program's behavior temporal aspect. In order to specify meaningful program behavior properties we have to enrich events with some attributes. An event may have a type and some other attributes, such as event duration, program source code related to the event, program state associated with the event (i.e. program variable values at the beginning and at the end of event), etc.

The next problem to be addressed after the program behavior model is set up is the formalism specifying properties of the program behavior. Since our goal is debugging automation, i.e. a kind of program dynamic analysis that requires different types of assertion checking, debugging queries, program execution profiles, and so on, we came up with the concept of a *computation over the event trace*. It seems that this concept is general enough to cover all the above mentioned needs in the unifying framework, and provides sufficient flexibility. This approach implies the design of a special programming language for computations over the event traces. We suggest a particular language called FORMAN [1], [3], [10] based on functional paradigm and the use of event patterns and aggregate operations over events.

Patterns describe the structure of events with context conditions. Program paths can be described by path expressions over events. All this makes it possible to write assertions not only about variable values at program points but also about data and control flows in the target program. Assertions can also be used as conditions in rules which describe debugging actions. For example, an error message is a typical action for a debugger or consistency checker. Thus, it is also possible to specify debugging strategies.

The notions of event and event type are powerful abstractions which make it possible to write assertions independent of any target program. Such generic assertions can be collected in standard libraries which represent the general knowledge about typical bugs and debugging strategies and could be designed and distributed as special software tools.

FORMAN is a general language to describe computations over program event trace that can be considered as an example of *a special programming paradigm*. Possible application areas include program testing and debugging, performance measurement and modeling, program profiling, program animation, program maintenance and program documentation [5]. A study of FORMAN application for parallel programming is presented in [4]

2 Events, Event Traces, and the Language for Computations Over Event Traces

FORMAN is based on a semantic model of target program behavior in which the program execution is represented by a set of events. An *event* occurs when some action is performed during the program execution process. For instance, a message is sent or received, a statement is executed, or some expression is evaluated. A particular action may be performed many times, but every execution of an action is denoted by a unique event.

Every event defines a time interval which has a beginning and an end. For atomic events, the beginning and end points of the time interval will be the same. All events used for assertion checking and other computations over event traces must be detectable by some implementation (e.g. by an appropriate target program instrumentation.) Attributes attached to events bring additional information about event context, such as current variable and expression values.

The model of target program behavior is formally defined through a set of general axioms about two basic relations, which may or may not hold between two arbitrary events: they may be sequentially ordered (PRECEDES), or one of them might be included in another composite event (IN). For each pair of events in the event trace no more than one of these relations can be established.

There are several general axioms that should be satisfied by any events a, b, c in the event trace of any target program.

1) Mutual exclusion of relations.

```
a PRECEDES b  => not (a IN b) and not (b IN a)
a IN b  =>  not(a PRECEDES b) and not (b PRECEDES a)
```

2) Noncommutativity.

```
a PRECEDES b  =>  not( b PRECEDES a)
a IN b  =>  not( b IN a)
```

3) Transitivity.

```
(a PRECEDES b ) and ( b PRECEDES c ) => ( a PRECEDES c)
(a IN b ) and ( b IN c ) => ( a IN c)
```

Irreflexivity for PRECEDES and IN follows from 2). Note that PRECEDES and IN are irreflexive partial orderings.

4) Distributivity

```
(a IN b) and (b PRECEDES c) => (a PRECEDES c)
(a PRECEDES b) and (c IN b) => (a PRECEDES c)
(FOR ALL a IN b (FOR ALL c IN d (a PRECEDES c) )) => (b PRECEDES d)
```

In order to define the behavior model for some target language, types of events are introduced. Each event belongs to one or more of predefined event types, which are induced by target language abstract syntax (e.g. execute-statement, send-message, receive-message) or by target language semantics (rendezvous, wait, put-message-in-queue).

The target program execution model is defined by an event grammar. The event may be a compound object and the grammar describes how the event is split into other event sequences or sets. For example, the event execute-assignment-statement contains a sequence of events evaluate-right-hand-part and execute-destination. The evaluate-right-hand-part, in turn, consists of an unique event evaluate-expression. The event grammar is a set of axioms that describe possible patterns of basic relations between events of different type in the program execution history, it is not intended to be used for parsing actual event trace.

The rule `A :: ( B C)` establishes that if an event a of the type A occurs in the trace of a program, it is necessary that events b and c of types B and C, also exist, such that the relations `b IN a, c IN a, b PRECEDES c` hold.

For example, the event grammar describing the semantics of a PASCAL subset may contain the following rules. The names, such as `execute-program,` and `ex-stmt` denote event types.

execute-program :: (ex-stmt *)

This means that each event of the type `execute-program` contains an ordered (w.r.t. relation PRECEDES) sequence of zero or more events of the type `ex-stmt`.

ex-stmt :: (label? (ex-assignment | ex-read-stmt | ex-write-stmt |

ex-reset-stmt | ex-rewrite-stmt | ex-close-stmt | ex-cond-stmt |

ex-loop-stmt | call-procedure))

The event of the type `ex-stmt` contains one of the events `ex-assignment, ex-read-stmt`, and so on. This inner event determines the particular type of statement executed and may be preceded by an optional event of the type `label` (traversing a label attached to the statement).

ex-assignment :: (ex-righthand-part destination)

The order of event occurrences reflects the semantics of the target language. When performing assignment statement first the right-hand part is evaluated and after this the destination event occurs (which denotes the assignment event itself). The event grammar makes FORMAN suitable for automatic source code instrumentation to detect all necessary events.

An event has attributes, for instance, source text fragment from the corresponding target program, current values of target program variables and expressions at the beginning and at the end of event, duration of the event, previous path (i.e. set of events preceding the event in the target program execution history), etc.

FORMAN supplies a means for writing assertions about events and event sequences and sets. These include quantifiers and other aggregate operations over events, e.g., sequence, bag and set constructors, boolean operations and operations of target language to write assertions on target program variables [2] [3]. Events can be described by patterns which capture the structure of event and context conditions. Program paths can be described by regular path expressions over events.

The main extension for the parallel case [4] consists of the introduction of a new kind of composite event -- "snapshot," which can be considered an abstraction for the notion "a set of events that may happen at the same time." The "snapshot" event is a set of events each pair of which is not under the relation PRECEDES, this makes it possible to describe and to detect at run-time such typical parallel processing faults as data races and deadlock states.

3 Examples of Debugging Rules and Queries

In general, a *debugging rule* performs some actions that may include computations over the target program execution history. The aim is to generate informative messages and to provide the user with some values obtained from the trace in order to detect and localize bugs. Rules can provide dialog to the user as well. An assertion is a boolean expression that may contain quantifiers and sequencing constraints over events.

Assertions can be used as conditions in the rules describing actions that can be performed if an assertion is satisfied or violated. A debugging rule has the form:

assertion **SAY** (expression sequence)

ONFAIL SAY (expression sequence)

The presence of metavariables in the assertion makes it possible to use FORMAN as a debugger query language. The computation of an assertion is interrupted when it becomes clear that the final value will be False, and the current values of metavariables can be used to generate readable and informative messages.

The following examples have been executed on our prototype FORMAN/PASCAL assertion checker [2], [3]. The

PASCAL program reads a sequence of integers from file XX.TXT.

```
program e1;
    var X: integer;
    XX: file of text;
begin
    X:= 7;
    (* initial value is assigned here *)
    reset (XX, 'XX.TXT');
    while X<>0 do
          read(XX, X)
end.
```

The contents of the file XX.TXT are as follows:

```
11 5 3 7 8 9 3 13 2 3 45 8 754 45567 0
```

Example of a Query 1. In order to obtain the history of variable X the following computation over event trace can be performed. The rule condition is TRUE, and is shown as a side effect the whole history of variable X.

```
TRUE
SAY ( 'The history of variable X is:'
    [D: destination  IS X FROM execute_program APPLY VALUE(D) ] )
```

The `[ ... ]` construct above defines a loop over the whole program execution trace (`execute_program` event). All events matching the pattern `destination IS X` are selected from the trace and the function VALUE is applied to them. The resulting sequence consists of values assigned to the X variable during the program execution.

When executed on our prototype the following output is produced:

```
Assertion #1 checked successfully...

The history of variable X is: 7 11 5 3 7 8 9 3 13 2 45 8 754 45567 0
```

***Example of an Assertion** 2.* Let's write and check the assertion : *"The value of* variable X does not exceed 17."

```
FOREACH *S: ex_stmt CONTAINS (D: destination IS X) FROM execute_program
```

```
VALUE(D) < 17

  ONFAIL

      SAY(‘Value ‘ VALUE(D) ‘is assigned to the variable X in stmt ‘)

      SAY(S)

      SAY(‘This is record #’ CARD[ ex_read_stmt FROM PREV_PATH(S)] + 1 ‘in the
file XX.TXT’)
```

We check the assertion for all events where the value of X may be altered. These are events of the type `destination` which can appear within `ex_assignment_stmt` or `ex_read_stmt` events. In order to make error messages about assertion violations more informative we include the embracing event of the type `ex_stmt`. Metavariables `S` and `D` refer to those events of interest. When the assertion is violated for the first time, the assertion evaluation terminates and current values of metavariables can be used for message output. The value of a metavariable when printed by the SAY clause is shown in the form:

```
event-type:> event-source-text

Time= event-begin-time .. event-end-time
```

Event begin and end times in this prototype implementation are simply values of step counter.

Since we expect the assertion might be violated when executing a Read statement, it makes sense to report the record number of the input file `xx.txt` where the assertion is violated. The program state does not contain any variables which values could provide this information. But we can perform auxiliary calculations independently from the target program using FORMAN aggregate operations. In this particular case the number of events of the type `ex_read_stmt` preceding the interruption moment is counted. This number plus 1 (since the violation occurs when the read statement is executed) yields the number of an input record on which the variable X was first assigned the value exceeding 17.

```
Assertion # 2 violation!

 Value 45 is assigned to the variable X in stmt

ex_stmt :> Read( XX , X )     Time= 73 .. 78

This is record # 11 in the file XX.TXT
```

Example of a Query 3. Profile measurement. In order to obtain the actual number of statements executed, the following query can be performed:

```
TRUE

SAY(‘The total number of statements executed is:’

    CARD[ ALL ex_stmt FROM execute_program ])
```

The `ALL` option in the aggregate operation indicates that all nested events of the type `ex_stmt` should be taken into account.

```
    Assertion #3 checked successfully...
 The total number of statements executed is: 18
```

Example of a *generic assertion* which must be true for any program in the target language.

“Each variable has to be assigned value before it is used in an expression evaluation.”

```
FOREACH * S: ex_stmt FROM execute_program
    FOREACH * E: eval_expression CONTAINS (V: variable) FROM S
          EXISTS D: destination FROM PREV_PATH(E) SOURCE_TEXT(D) = SOURCE_TEXT(V)
ONFAIL
    SAY( 'In event' S)
    SAY( 'in expression evaluation')
    SAY(E)
    SAY('uninitialized variable' SOURCE_TEXT(V) 'is used')
```

For the following PASCAL program our prototype detects the presence of the bug described above.

```
program e2;
var X,Y: integer;
begin     Y:= 3;
    if Y < 2 then begin
          X:= 7; Y:= Y + X
    else Y:= X - Y (*** here the error appears: X has no value! ***)
end.
```

```
Assertion #4 violation!
In event ex_stmt :> If ( Y < 2 ) then X := 7 ; Y := ( Y + X ) ;
                                      else Y := ( X - Y ) ; Time= 10 .. 35
in expression evaluation
eval_expression :> ( X - Y ) Time= 20 .. 29
uninitialised variable X is used
```

Debugging rules can be considered as a way of formalizing reasoning about the target program execution -- humans often use similar patterns for reasoning when debugging programs. For example, if the index expression of an array element is out of the range, the debugger can try a rule for eval-index events that invokes another rule about wrong value of the event eval-expression, which in turn will cause investigation of histories of all variables included in the expression.

Yet another application of generic assertions and debugging rules may be for describing run-time constraints (sequences of procedure calls, actual parameter dependences, etc.) for nontrivial subroutine packages, e.g. for the MOTIF package for GUI design. A library containing assertions and debugging rules relevant to such a package may be useful for writing C programs calling subroutines from the package.

4 Conclusions

In brief, our approach can be explained as "computations over a target program event trace." We expect the advantages of our approach to be the following:

- The notion of **an event grammar** provides a general basis for program behavior models. In contrast with previous approaches, the **event** is not a point in the trace but an interval with a beginning and an end.
- Event grammar provides a coordinate system to refer to any interesting event in the execution history. Program variable values are attributes of an event's beginning and end. Event attributes provide complete **access to each target program's execution state**. Assertions about particular execution states as well as assertions about sets of different execution states may be checked.
- The PRECEDES relation yields a **partial order** on the set of events, which is a natural model for parallel program behavior.
- The IN relation yields a **hierarchy of events**, so the assertions can be defined at an appropriate level of granularity.
- A language for **computations over event traces** provides a **uniform framework** for assertion checking, profiles, debugging queries, and performance measurements.
- The access to the complete target program execution history and the ability to formalize **generic assertions** can be used in order to define **debugging rules and strategies.**
- The fact that assertions and other computations over target program event trace can be **separated from the text of the target program** allows accumulation of formalized knowledge about particular programs and about the whole target language in separate files. This makes it easy to control the amount of assertions to be checked.

According to [7] and [12] approximately 40-50% of all bugs detected during the program testing are logic, structural, and functionality bugs, i.e. bugs which could be detected by appropriate assertion checking similar to the demonstrated above.

References

[1] M. Auguston, "FORMAN -- A Program Formal Annotation Language", *Proceedings of the 5:th Israel Conference on Computer Systems and Software Engineering*, Gerclia, May 1991, IEEE Computer Society Press, 149-154.

[2] M. Auguston, "A language for debugging automation", *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering,* Jurmala, June 1994, Knowledge Systems Institute, pp. 108-115.

[3] M. Auguston, "Program Behavior Model Based on Event Grammar and its Application for Debugging Automation", *in Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging,* Saint-Malo, France, May 1995.

[4] M. Auguston, P. Fritzson, "PARFORMAN -- an Assertion Language for Specifying Behavior when Debugging Parallel Applications", *International Journal of Software Engineering and Knowledge Engineering*, vol.6, No 4, 1996, pp. 609-640.

[5] M. Auguston, A. Gates, M. Lujan, "Defining a program Behavior Model for Dynamic Analyzers", *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering,* SEKE'97, Madrid, Spain, June 1997, pp. 257-262

[6] P. C. Bates, J. C. Wileden, "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach", *The Journal of Systems and Software* 3, 1983, pp. 255-264.

[7] B. Beizer, Software Testing Techniques, Second Edition, International Thomson Computer Press, 1990.

[8] F. Brooks, The Mythical Man-Month, 2nd edition, Addison-Wesley, 1995.

[9] B. Bruegge, P. Hibbard, "Generalized Path Expressions: A High-Level Debugging Mechanism", *The Journal of Systems and Software* 3, 1983, pp. 265-276.

[10] P. Fritzson, M. Auguston, N. Shahmehri, "Using Assertions in Declarative and Operational Models for Automated Debugging", *The Journal of Systems and Software* 25, 1994, pp. 223-239.

[11] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, vol. 21, No. 7, July 1978, pp. 558-565.

[12] S. L. Pfleeger, Software Engineering, Theory and Practice, Prentice Hall, 1998.

Performance Evaluation of Software Architectural Types: A Process Algebraic Approach

Marco Bernardo[1], Paolo Ciancarini[2], Lorenzo Donatiello[2]

[1]Università di Torino, Dipartimento di Informatica
Corso Svizzera 185, 10149 Torino, Italy
E-mail: bernardo@di.unito.it
[2]Università di Bologna, Dipartimento di Scienze dell'Informazione
Mura Anteo Zamboni 7, 40127 Bologna, Italy
E-mail: { cianca, donat }@cs.unibo.it

The software architecture level of design allows to cope with the increasing size and complexity of software systems during the early stage of their development [7, 8]. To achieve this, the focus is turned from algorithmic and data structure related issues to the overall architecture of the system. The architecture is meant to be a collection of computational components together with a description of their connectors, i.e. the interactions between these components.

As software architecture emerges as a discipline within software engineering, it becomes increasingly important to support architectural development with languages and tools. It is widely recognized that suitable architectural description languages (ADLs for short) should be devised to formalize software architectures instead of using informal box-and-line diagrams, and related tools should be implemented to support the automatic analysis of architectural properties in order to allow the designer to make principled choices.

As far as we know, almost all the existing languages and tools deal only with functional aspects of software architectures. However, designers are often faced with the problem of choosing among different software architectures which are functionally equivalent. This choice is thus driven by nonfunctional factors, and mostly by performance requirements. In general, as recognized in [9], performance analysis should be integrated into the software development process, starting from the earliest stages and continuing throughout the whole life cycle.

In order to create a framework where the functional and performance properties of formally represented software systems can be automatically evaluated at the architectural level, we need a suitable theory which provides the necessary underpinnings to the architectural concepts of component and connector. This is witnessed e.g. by the fact that a well known ADL like WRIGHT [1] is based on CSP [5].

In the field of computer and communication system modeling and analysis, several formal description techniques have been proposed in the last two decades which account for both functional and nonfunctional aspects of systems. Among such formal description techniques, there are stochastically timed process algebras (see, e.g., [2]). Their key feature is compositionality. First of all, like classical process algebras, they allow for compositional model construction because they are algebraic languages endowed with a small set of powerful operators, such as parallel composition, sequential composition, and alternative composition, which allow descriptions to be systematically built from their components. Unlike classical process algebras, stochastically timed process algebras come equipped with the capability of expressing activity durations by means of exponentially distributed random variables, so that the underlying performance models turn out to be Markov chains which can thus be exploited to effectively derive performance measures. Second, stochastically timed process algebras allow for compositional model manipulation. This is achieved by means of equivalences which relate terms possessing the same functional and performance properties. Whenever such equivalences are congruences, i.e. they are substitutive w.r.t. the algebraic operators, they permit to replace algebraic components with equivalent (smaller) ones without altering the overall system properties. Third, stochastically timed process algebras allow for compositional model solution whenever the underlying Markov chain meets certain conditions.

Since the compositionality of stochastically timed process algebras seems to be well suited for the architectural level of design, we propose their adoption for the development of an ADL aiming at predicting the performance of software systems and comparing the performance of several software architectures designed for the same system. Such an ADL has been called ÆMPA [3] as it is based on the process algebra EMPA [2].

A description in ÆMPA is an architectural type:

```
archi_type              ⟨name and parameters⟩
   archi_cc_types       ⟨architectural component/connector types⟩
   archi_topology       ⟨instances and attachments⟩
   archi_interactions   ⟨architectural interactions⟩
end
```

Each architectural type is defined as a function of its component and connector types, its topology, its interactions, and its generic parameters. A component/connector type is in turn defined as a function of its behavior, specified as a family of EMPA sequential terms, and its interactions, specified as a set of EMPA action types. The architectural topology consists of a set of component/connector instances related by a set of attachments among their interactions. Architectural interactions are interactions of component instances which support hierarchical architectural modeling. Finally, generic parameters are basically values for parametric rates and weights.

For the sake of ease of use, the textual notation above is accompanied by a graphical notation inspired by the flow graphs of [6], as they may provide a visual help to the development of architectural descriptions. A flow graph is a network of nodes each equipped with a set of ports; two ports of two different nodes are linked together if the two nodes can interact. Given an ÆMPA description, each component/connector type can be represented as a node (depicted as a rounded box) with its behavior textually reported inside the node and its interactions labeling the ports (depicted as black circles, or white squares in the case they refer to the whole architecture). We then create an instance of a node for each instance of the component/connector type it refers to. Finally, nodes are linked together according to the specified attachments. In case of hierarchical architectural modeling, a node can contain a flow graph whose architectural interactions are linked (through dashed lines) to the ports of the node.

The concept of architectural type, proposed in [4], is an abstraction of the concept of architectural style. Given an architectural style, the set of component and connector instances and their internal behavior can vary from architectural instance to architectural instance, but the structure of the overall interconnection of component and connector instances and their internal behavior w.r.t. interactions is fixed. Because of the presence of two degrees of freedom (variability of the set of component and connector instances and variability of their internal behavior), investigating the properties which are common to all the instances of an architectural style is not an easy task. To make such a task manageable, architectural types are advocated in [4] because they constrain the set of component and connector instances to be fixed. The instances of a given architectural type are then generated by letting the behavior of component and connector types vary. In other words, the component/connector types specified in an architectural type are viewed as being formal, so one can call for an architectural type and pass to it actual component/connector types.

The formal semantics for ÆMPA is given by translation into EMPA by essentially exploiting the parallel composition operator. Given that the semantics of a component/connector type is the family of EMPA sequential terms expressing its behavior and that the semantics of a component/connector instance is the semantics of the related type, the semantics of an architectural type is obtained by composing in parallel the semantics of the component/connector instances according to the specified attachments. Once the corresponding family of EMPA terms has been generated, the analysis of the architectural properties can be carried out by means of existing tools. In particular, it is possible to verify functional properties (like deadlock freedom and mutually exclusive use of resources) through model checking, equivalence checking, and preorder checking. Likewise, it is possible to evaluate performance measures (such as system throughput and user response time) via exact or approximate Markovian analysis or simulation.

We observe that, from the process algebra perspective, creating an ADL can be viewed as an attempt to force the designer to model systems in a more controlled way, which in particular elucidates the basic architectural concepts of component and connector and hopefully enhances the usability of process algebras. However, this syntactic sugar alone is not enough to create a useful ADL. It must be accompanied by suitable techniques to verify the well formedness of architectural descriptions, such as the architectural compatibility and conformity checking. The purpose of the former check is to ensure that an architectural type is well connected, in the sense that every pair composed of a component instance and a connector instance attached to each other interact in a proper way. This is formalized by requiring that the functional behavior of the two instances, when projected on the interactions involved in the related attachments, is the same. The latter check, instead, aims at guaranteeing that the actual parameters are consistent with the formal ones in case of architectural type invocation. This is formalized by requiring that the actual parameters do not alter the functional semantics of the architectural type w.r.t. component and connector interactions, i.e. the functional behavior of the architectural type when projected on such interactions. Technically, both checks are carried out by means of the weak bisimulation equivalence [6], a purely functional equivalence whose major feature is its ability to reason about the functional behavior of process terms when projected on certain actions, i.e. when abstracting from unimportant actions. It is worth observing that the architectural conformity checking can be compositionally conducted parameter by parameter as the weak bisimulation equivalence is a congruence w.r.t. the parallel composition operator.

We conclude by mentioning the fact that we are now in the process of implementing a software tool for the functional and performance analysis of well formed architectural types specified with the textual or graphical notation of ÆMPA. Such a tool will rely on the EMPA based software tool TwoTowers [2] and will allow us to conduct some case studies to assess the adequacy of our approach. For the time being, we are using a prototype of the tool to investigate the properties of several load distribution algorithms for replicated web services in different architectural scenarios.

Acknowledgements

This research has been funded by Progetto MURST Cofinanziato SALADIN: *"Architetture Software e Linguaggi per Coordinare Componenti Distribuite e Mobili"*.

References

[1] R. Allen, D. Garlan, *"A Formal Basis for Architectural Connection"*, in ACM Trans. on Software Engineering and Methodology 6:312-249, 1997

[2] M. Bernardo, *"Theory and Application of Extended Markovian Process Algebra"*, Ph.D. Thesis, University of Bologna (Italy), 1999 (`http://www.di.unito.it/~bernardo/`)

[3] M. Bernardo, P. Ciancarini, L. Donatiello, *"ÆMPA: A Process Algebraic Description Language for the Performance Analysis of Software Architectures"*, to appear in Proc. of the *2nd ACM Int. Workshop on Software and Performance (WOSP 2000)*, ACM Press, Ottawa (Canada), 2000

[4] M. Bernardo, P. Ciancarini, L. Donatiello, *"On the Formalization of Architectural Types with Process Algebras"*, to appear in Proc. of the *8th ACM Int. Symp. on the Foundations of Software Engineering (FSE-8)*, ACM Press, San Diego (CA), 2000

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

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

[7] D.E. Perry, A.L. Wolf, *"Foundations for the Study of Software Architecture"*, in ACM SIGSOFT Software Engineering Notes 17:40-52, 1992

[8] M. Shaw, D. Garlan, *"Software Architecture: Perspectives on an Emerging Discpline"*, Prentice Hall, 1996

[9] C.U. Smith, *"Performance Engineering of Software Systems"*, Addison-Wesley, 1990

Appliances and Software: The Importance of the Buyer's Warranty and the Developer's Liability in Promoting the Use of Systematic Quality Assurance and Formal Methods

Daniel M. Berry

Computer Science Department
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada

dberry@csg.uwaterloo.ca

Abstract

A vexing question is why systematic quality assurance and formal methods, despite all their advantages, are not used routinely in software development. Other engineering disciplines use their systematic quality assurance and formal methods in producing routinely reliable products. Perhaps the difference between the other engineerings and software engineers lie in the warranties producers must give for the products and the liabilities suffered by the producers for malfunctioning products. It is argued that software engineers would be more likely to use formal methods more routinely if they or their employers had to guarantee their software and could be sued for damages caused by their malfunctioning software.

1 Introduction

Formal methodologists continue to bemoan the failure of practicing software engineers to employ formal methods in their daily software development work [15, 30, 17, 8, 9, 12, 22, 23, 21, 11]. Early attempts by formal methodologists to convince software engineers to use formal methods focused on the benefits to software quality that would accrue if formal methods were to be used regularly in software development [20, 14, 11, 12]. Surely, once a software practitioner understood the benefits, he or she would start to use formal methods enthusiastically. To fail to do so would be illogical! Illogical or not, software engineers by and large, ignored formal methods, and when forced to use formal methods, they resist and sometimes actively subvert the application of formal methods to do meaningless busy work. When the project fails, partially due to the subversion, they gleefully blame the formal methods for the failure. Successful experimental applications of formal methods to real projects [21, 18, 10, 7, 13, 1, 2, 5, 31, 3, 19, 4] failed to convince most software engineers of the effectiveness of formal methods. The perception is that the project team got lucky or had other strengths going for it or that the project had special security or safety needs that could not be handled with ordinary methods, needs that are not found in normal software.

Formal methodologists began exploring ways to make formal methods more attractive. Most of these were technical solutions aimed at making the formal language, the method, the tools, etc. more palatable, more easily used, more powerful, more automatic, less ambitious, more realistic, more incremental, and even more fun [18, 10, 7, 13, 11, 1, 2, 5, 31, 3, 19, 4]. Each paper about one of these new approaches bemoans the lack of general use of formal methods, diagnoses the lack as the result of some particular problem in the use of formal methods, and offers a new approach that avoids, mitigates, or solves the identified problem. However, none of these has had any real effect on the extent to which formal methods are used. Others try educational, sociological, and managerial approaches [12, 23, 26, 6], and they too have failed to produce the desired bandwagon.

Even non-formal, but systematic methods for ensuring software quality suffer the same lack of use. Advocates of software inspection [16, 29] note the empirical evidence of inspection's effectiveness at finding faults before execution, an effectiveness exceeding that of traditional testing. They also note the reluctance of many organizations to use inspection and the varied, creative excuses given for not using inspection, even when they know its effectiveness.

Yet, when we look at other engineering disciplines, we see that they all have their systematic quality assurance (QA) and formal methods. Civil engineering has mathematical models of load and stress and these allow calculating, on the basis of only a paper design for a bridge, whether the proposed bridge will support the required weight, and then some. Electrical engineering has mathematical models of circuitry that allow calculating, on the basis of only a circuit diagram, whether the proposed circuit will behave as required, will not overheat, etc. We see that engineers in these disciplines routinely apply their formal methods with no complaints of being overburdened with useless work, no complaints of their creativity being stifled, and no complaints of having to use the dull, dreaded *mathematics* in another field.

The questions to ask are:

1. what makes the engineers in the other, more traditional engineerings, apply their systematic QA and formal methods routinely, and

2. can whatever does the trick in these other engineerings be used to get software engineers to use software engineering's systematic QA and formal methods in their daily software development?

This paper explores the quality of different engineering products, some electromechanical, some electronic, and some software, and notes key differences in the warranties offered with these products and the liabilities borne by their developers. Perhaps these differences account for the differences in the willingness of the various engineers to apply their engineerings' systematic QA and formal methods.

2 Recent Experience with Purchased Appliances and Software

In the last two years (as of November, 1999), I have bought* four appliances and four pieces of software. I am still using all the appliances but I have yet to get the two of the programs running, one of these is gathering dust on my shelf and the other has been returned for a refund. The other two programs are working. The four appliances are

1. Sharp Carousel Microwave Oven,

2. RCA Color Television,

3. Toshiba Video Cassette Recorder, and

4. Hoover Futura Vacuum Cleaner.

The four programs are

1. Adobe Illustrator 7.0,

2. Adobe Acrobat Exchange 3.0,

3. Microsoft Office '97, and

4. Languageforce Deluxe Universal Translator.

These eight personal experiences amount to a case study giving anecdotal evidence in support of a popular perception that consumer software is of considerably poorer quality than consumer appliances.

* Strictly speaking, one does not buy software; he or she licenses software. However, common usage is "to buy software".

Observe that all the software in this list is developed by their producers for sale to the mass market of consumers and is different from *bespoke* software developed by one producer under a specific negotiated contract for a specific client.

2.1 Appliances

I assembled the vacuum cleaner according to directions provided in the package and then plugged it in. It worked immediately and has given no problem. I would quibble with the design of the cord wrap-around knobs. It is impossible to turn the machine on when any of the wire is wrapped around the knobs since the on-off switch doubles as a knob. However, I have no real complaints about the machine or its brief instructions.

To get the microwave oven working, I had to set a timer according to directions provided in the package. It took less than 10 minutes to read the entire manual to see what features it had and to decide which ones I would use. I set the clock with no problem and the oven has worked fine ever since. Unfortunately, the oven has no battery back up. Consequently, whenever there is a power hiccup, the oven resets and I have to reset the timer to get the overn to work again. Apart from this minor nuisance, I have no complaints about the oven or its documentation.

The television required a set up. The procedure was well explained, step-by-step, in the user's manual, and the set worked immediately. The set-up procedure had as its main goal to allow the television set to find all the channels in the antenna, cable, or cable box connected to the set. In my current situation, I had cable, and the set found some 50 channels. Two years earlier, in another country, I had bought a television to use with a cable box. That television set was also an RCA. Since the cable box mapped all cable channels to television channel 3, the set-up was trivial, and the television worked immediately.

Among the appliances, the video cassette recorder offered the most trouble. The manual seemed quite clear in that I could read step-by-step procedures, and I could see what functions I wanted to use. What I did not know was that the manual neglected to mention that it was necessary to undo one set up if you want to change to a different set up. The set up has as its purpose to find all the available channels coming in from the antenna, cable, or cable box. It went smoothly according to what the manual appeared to say. The record and play worked immediately. When I tried to program my first timed record, I saw that I had overlooked an important step that is to be done before the set up and that I had done the wrong set up. I did the important step, and the correct set up, and then tried to program a timed record again. It still did not work. Fortunately, I had paid for a warranty that provided in-house service. I called and arranged for service. At first, they wanted me to bring the set in. I told them that I had paid for in-house service. They said that it is better if I bring it in. I said that if they did not honor the service agreement immediately, I would return the set for a refund and tell everyone at the University of Waterloo not to shop at the store to whose service department I was speaking. They sent a technician out who walked through the set up with me after telling me that I had to undo the first set up. It seems that without undoing the first set up, I was leaving elements of the old set up in the computer and it was creating an inconsistent program that caused the whole programmed record to freeze. After the technician left, the programmed record worked perfectly and I have had no complaints since then. I did offer as a project in my graduate requirements engineering seminar the job of rewriting the user's manual.

2.2 Software

Contrast these appliance experiences with my software experiences. The difference is sobering.

I was still in Israel when I bought the copy of Adobe Illustrator 7.0 for Windows from a local supplier. Normally, I like to read the manual for software before doing any installation both to see ahead of time where the problems might be and to learn what features I was likely to use. I verified that my computer had more than the minimum of all the resources required. I followed the installation procedure described in the manual supplied with the software, doing what the install program asked me to do and answering its questions about the features I wanted. In answering these, I tended to include a feature that I was not sure about just to be sure that I got a minimal working set of features. The installer reported that Illustrator was successfully installed. I tried to start up the program, but it froze during its start up phase, never getting beyond the product name announcement screen. I figured that I had too many other programs running and killed them all. I started up Illustrator again and it froze again. Each time it froze, I could not get the computer and Windows unstuck and had to turn the machine off and then on without going through the proper shut down procedure. I called the local supplier and was told that this program and other Adobe products do not work on the Hebrew version of Windows or even the Enabler version that allowed switching between Hebrew and English. I had to use a pure American version of Windows. I was upset at the local supplier for not telling me this little detail when I bought the program.

I changed my operating system to pure American Windows. Since I did all my work in English, it was no real problem not to have Hebrew or the ability to switch to Hebrew. I could use a departmental machine in the very rare occasion in which I needed Hebrew Windows. Installation appeared to succeed. Illustrator, however, still froze when I tried to run it, even when no other programs were running. Figuring that maybe my PC was too small, I put the software away for my planned move to Canada, at which time I would get a larger PC.

Once settled at the University of Waterloo in Canada, I arranged for a large machine. Of course the Windows on it was purely American. I installed Illustrator on that machine. Illustrator still froze the machine as it started up. I ended up deinstalling the program only after copying the large set of POSTSCRIPT Type 1 fonts Illustrator 7.0 provides; I decided to install them for use with troff on my UNIX platform. By this time it was too late to return the software for a refund.

When the software had failed to work, I tried to get help from Adobe. I could not find in the list of reported problems at the web site any solutions that I had not tried. Adobe provided an 800 phone number to call customer service. However, from overseas, 800 numbers are not free. Also, I do not hear well on the phone. I could find no button that sends e-mail for help. So, I sent e-mail to support@adobe.com, webmaster@www.adobe.com, and postmaster@www.adobe.com, explaining that I could not find my problem solved at their web site and I cannot hear on the telephone. I got no reply. The program sits unused on my bookshelf. I did get a nice set of fonts out of it, and Illustrator costs less than the sum of the costs of the individual fonts, but I feel that I threw out money, and I have no operating Illustrator. In all fairness, it is not clear where the fault lies; the problem could be in the Windows operating system on which I tried to run Illustrator.

I bought a copy of Adobe Acrobat Exchange 3.0 for Windows. This software comes with a hard copy manual describing only installation. The general user's manual is a PDF document readable by the software once the software is installed. The Acrobat Exchange 3.0 package consists mainly of two programs, AcroExch and Distiller, which manipulate PDF files and create PDF files from POSTSCRIPT files, respectively. They installed with no problem. I was able to print a hard copy of the user's manual. I read it and decided what features I wanted to use and how. I have been using both to prepare slide shows of the troff-generated POSTSCRIPT of lecture slides. I use Distiller to convert the troff-generated POSTSCRIPT into PDF and then I use AcroExch to rotate and crop pages and to make hypertext links between items in the slides for faster navigation during a lecture. Given my past experience with an Adobe product, it was a pleasant surprise that this Adobe product installed and works so well. I wish that all mass-produced software worked as well.

Even as I write this paper, the system guru for my Sun workstation is not able to get the publically downloadable Adobe Acrobat Reader 4.0 working on my workstation. It freezes up at the start up window. Fortunately it does not freeze up the machine. A control-C to the invoking window kills the program. Again, I am unable to find my problem discussed in the trouble shooting section of Adobe's web pages, Adobe has shut down the user's forum, and they do not answer my e-mail cries for help even when I explain that I cannot use the telephone.

I normally do not use any program in Office '97. I use troff for formatting, vi for editing, troff and Acrobat Distiller for slide shows, and vi and awk for spread sheeting. However, people insist on sending me Word, PowerPoint, and Excel documents. So I bought Office '97. I bemoaned the lack of a real hard-copy manual; the manual provided with Office '97 consists only of a tutorial for each program in the collection. I installed the software. The installation went well, the only problem being a lack of good information about what features I really needed. The programs seem to work well on 95% of the Word, PowerPoint, and Excel documents I receive. However, occasionally I get a document which when printed leaves a lot of characters unprinted. I see each of these characters on the screen, but on the paper, there is only white space in place of the character. I found a section of Help that described this problem. It said that the problem was that the fonts selected for the missing text is not resident in the printer and that the solution was to request downloading of all fonts. Unfortunately, when I went to the right menu to request full downloading, I found that the option was already in force. An examination of the POSTSCRIPT file generated by Word showed that definitions for all the selected fonts were included in the file and that the missing text was not even in the POSTSCRIPT file. No wonder this text did not print! Occasionally, I ended up printing an image of the Word window to get hard-copy output. The most annoying thing about Word, apart from its very deficient editing capability and its abominable typesetting with irregular spacing between words and lack of ligatures, is the fact that I am asked when I am exiting Word, if I want to update the changes to I made to PDFWRITE.DOT. Of course not, since I did not make any changes to it as far as I am concerned, having done nothing with PDF in the document. However, once I answered too quickly and clicked the "Yes" button. Word would not work properly after that. I had to reinstall Word from the beginning. It has happened several times, when I have accidentally hit the wrong button and I had to reinstall Word each time.

I get documents in languages other than English, e.g., in Hebrew, French, Spanish, Portuguese, and German. I can read these, but occasionally need a dictionary to fill in holes in my understanding. When I saw this new program, Languageforce's Universal Translator, that offered multiple language dictionary functionality, I decided to buy it. It offered a way to avoid the necessity to keep a half dozen dictionaries on hand and to avoid the tedium of flipping pages to find a word. At first, I was really happy about the software. It had a real user's manual. It consisted of a set of scenarios for set up and various different ways of using the software, including as a dictionary. The manual even showed pictures of the screen that the user is supposed to receive after various steps in the scenarios. However, this joy was short lived. As I was following the set up procedure, I saw that a number of the screen pictures were either wrong or out of date. In any case, the installation program reported a successful installation. Then I tried to run the program. It simply would not run. It gave no explanation. I sent e-mail to Languageforce's support group to the address given in the user's manual. I never got *any* reply. However, I know that someone received the e-mail, because to this day, I get announcements of exciting new enhancements from the sales department. Even a message from me telling them that they have some nerve offering enhancements to me when they have not answered my request for help has failed to get these messages turned off. Needless to say, I took the program back for a full refund.

2.3 Software Released Too Early

After these experiences, I started to wonder what can be done to improve the use of quality assurance methods in the development of consumer software. After all, methods and technology do exist to do a better job with software. However, they are not being used in the rush to get software out to the market. In this rush, it appears that software is being released before it is ready. Software is going out for sale to consumers before it is certain that it will run and with the documentation woefully inadequate and even incorrect. Moreover, the manufacturers seem unprepared and even unwilling to service their shoddy merchandise. It might even be that the merchandise is so shoddy that the service people are overwhelmed and the shoddy service is a direct result of this overload. On the other hand, appliances for sale generally work with no trouble and continue to work and when they need service, the manufacturers stand behind the product and do service the products in a reasonable time. Once serviced, the problems seem to be solved.

A major reason that software is released before it is ready is the pressure on the producer to be first on the market with the product. Whoever is first usually gets and keeps a vast majority of the market. The second to the market usually gets very little market and fails as a business unless its product is perceived as at least an order of magnitude better than that of the first.* Therefore, there is a high incentive to release software early. Moreover, since customers accept shoddy software merchandise, there is very little incentive to delay release to improve the product. Any solution to this problem will have to reverse the incentives.

* Of course, there are the exceptions that make the rule. For example Apple's Macintosh system beat Microsoft's Windows system to the market by several years, but Microsoft nearly drove Apple out of business.

3 Analysis of Differences Between Software and Appliance Productions

What are the differences between appliances and software that might account for this observed difference in quality?

Certainly software is more complex and has more states than do vacuum cleaners. However, a television and a video cassette recorder are systems of moderate complexity matching that of many programs. Indeed, these machines probably implement some of their functionality with a computer and software.

Certainly the environment on which software runs is more varied than that on which appliances run. Software must run on a variety of CPUs and operating systems and flavors thereof. Appliance environments are far simpler, consisting of an electrical outlet and the television signals coming through a cable wire or over the air. However, in my case, all my PCs had very standard configurations. I bought them strictly for use of the Office programs that were not available on the Sun workstation, on which I do all my real work. I left them in their original, delivered, presumably standard, configurations. I would have expected the software to have been tested for running on my configurations.

In my opinion, one key difference is the difference in the warranty that comes with appliances and with software. An appliance is forced by law in most locales in the U.S. and Canada to have a warranty of fitness for its purpose. That is, the product is guaranteed to function as what it is. If I buy a television set, the manufacturer guarantees that it functions as a television set and as a television set as understood by the man in the street. Mass-produced software, however, traditionally comes with a shrinkwrapped license that says that the manufacturer warrants almost nothing about the behavior of the software. The manufacturer does warrant the medium on which one buys the software, the diskettes or the CD ROM. In other words, the manufacturer refuses to guarantee that Illustrator program actually allows the user to draw pictures, that Word actually formats documents, that PowerPoint actually makes slide shows, and that Universal Translator actually translates. The software manufacturers refuse to make these guarantees, because they are not required to by law, as are appliance manufacturers, and the software customers let them get away with it. What manufacturers are not required to do, they do not do, and the customers suffer.

Another key difference is the difference in liability borne by the producers of appliances and software. Appliance manufacturers are liable for damages caused by correctly used or malfunctioning appliances. Software producers disclaim almost all liability in their shrinkwrapped licenses, accepting liability only up to the cost of the software. Thus, software developers do not have to be as careful with their mass-market products as appliance manufacturers do.

3.1 Warranties

This section examines some warranties supplied with the software products and appliances described in Section 2. Each warranty's text is quoted in a sans serif font, allowing the quotation to be distinguished from comments that interrupt the quotation just after the sentences they discuss.

3.1.1 Software Warranties

Adobe's and Microsoft's End User License Agreement (EULA) are almost identical. Therefore, only one is quoted here. Adobe's EULA says:

> **5. Limited Warranty.** Adobe warrants to you that the Software will perform substantially in accordance with the Documentation

Who gets to decide how large the deviation from the documentation-described behavior is allowed and the software is still considered to perform *substantially* in accordance with the documentation?

> for the ninety (90) day period following your receipt of the Software.

The warranty applies for only 90 days, as if the software decays and might start to perform differently after 90 days. It sounds like this warranty provision was written to get mostly computer-and-software-illiterate legislators off the producer's back and mimics the warranty given with products that can decay and go bad after a period of use. Ninety days would seem a reasonable amount for the producer to stand behind such a decaying product. I am quite sure that some computer-and-software-illiterate people think that software decays.

> [missing details dealing with fonts that are translated to other formats; the warranty does not apply to these other formats.]
>
> To make a warranty claim, you must return the Software to the location where you obtained it along with a copy of your sales receipt within such ninety (90) day period. If the Software does not perform substantially in accordance with the Documentation, the entire and exclusive liability and remedy shall be limited to either, at Adobe's option, the replacement of the Software or the return of the license fee you paid for the Software.

The producer gets to choose the remedy, not the customer. Moreover, The producer is permitted to replace the software, as if a different copy of the software will behave differently. Here again, it sounds like this provision is written to get naive legislators off the producer's back with something that makes it appear that the producer is really trying to get to the user something that works. I hope that the customer can force the supplier to adopt the money-back remedy when it is clear that another copy of the same program is going to perform exactly the same as the non-conforming copy.

> ADOBE AND ITS SUPPLIERS DO NOT AND CANNOT WARRANT THE PERFORMANCE OR RESULTS YOU MAY OBTAIN BY USING THE SOFTWARE OR DOCUMENTATION. THE FORGOING STATES THE SOLE AND EXCLUSIVE REMEDIES FOR ADOBE'S OR ITS SUPPLIER'S BREACH OF WARRANTY. EXCEPT FOR THE FORGOING LIMITED WARRANTY, ADOBE AND ITS SUPPLIERS MAKE NO WARRANTIES, EXPRESS OR IMPLIED, AS TO NONINFRINGEMENT OF THIRD PARTY RIGHTS, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE.

In any case, the producer denies, and shouts this denial of, any warranty for what really matters, in particular for merchantability and for fitness for *any* purpose, including for the very purpose of the software. Note that *all* non-software products are warranted by law in the U.S. and Canada for merchantability and for fitness for their stated and implied purposes.

> Some states or jurisdictions do not allow the exclusion of implied warranties or limitations on how long an implied warranty may last, so the above limitations may not apply to you. To the extent permissible, any implied warranties are limited to ninety (90) days. This warranty gives you specific legal rights. You may have other rights which vary from state to state or jurisdiction to jurisdiction. For further warranty information, please contact Adobe's Customer Support Department.

The reality is that very few places do not allow exclusion of implied warranty for software products.

Elsewhere in the license, it is written:

> This package contains software ("Software") and related explanatory written materials ("Documentation").

Adobe Illustrator 7.0 and Microsoft Office '97 come with reasonably good, descriptive manuals describing some typical scenarios the users might wish to do. Therefore, it might appear that the software is being warranted to behave as the manual says it does. However, the warranty specifies only *substantial* compliance with the written documentation, not complete compliance. Who decides how much compliance is substantial enough? In addition, it might be that the software can do *all* the scenarios that are described in the manual, as these were the test cases. Certainly the developer had to get these examples running to get the pictures of the screen that are shown in the manual. However, the software does nothing more general, because the manual describes *all* the test cases. In other words, the documentation means only what it says and not what the average reader generalizes it to say.

The only written material I find in many packages these days is a manual describing only installation. Given the typical EULA as described above, perhaps the producer is warranting only that the installation, and not necessarily the program, will perform substantially, but not necessarily completely, in accordance with the documentation provided. Of course, there is the help system providing documentation, but if the software does not run, and the help system does not work, does that mean that the software is effectively not documented or that if the user cannot get to the documentation, any behavior is allowed for the software because it is undefined in the documentation?

Clearly, the warranty accompanying software is next to useless except for getting one's money back if the software does not work.

3.1.2 Appliance Warranties

The Hoover vacuum cleaner comes with a warranty that says:

> **Full One Year Warranty** (Domestic Use)
>
> Your HOOVER® appliance is warranted in normal household use, in accordance with the Owner's Manual against original defects in material and workmanship

The warranty covers all defects in what comes from the manufacturer. There is no concept of performing only substantially as a vacuum cleaner. The appliance must perform completely as a vacuum cleaner from the beginning.

> for a period of one full year from date of purchase. This warranty provides, at no cost to you, all labor and parts to place this appliance in correct operating condition during the warranted period.

Hoover is saying that it *will* make the appliance work, by replacing whatever is necessary and doing whatever work is necessary to get it running. Presumably, Hoover could replace all parts together as a unit, that is, provide a whole new vacuum cleaner.

> This warranty applies when the appliance is purchased in the United States including its territories and possessions, or in Canada, or from a U. S. Military Exchange. Appliances purchased elsewhere are covered by a limited one year warranty that covers the cost of parts only.

While a customer might not be covered for labor costs outside the U.S. and Canada, the parts and presumably a replacement are covered. Also, if Hoover has built the vacuum cleaner well enough to be sold in the U.S. and Canada, the vacuum cleaner will in all probability not need any such repair. Thus, the lack of a full warranty is not disturbing.

> This warranty does not apply if the appliance is used in a commercial or rental application.

Hoover is covering only normal household use, not heavy-duty use.

> Warranty service can only [sic] be obtained by presenting the appliance to one of the following authorized warranty service outlets. Proof of purchase will be required before service is rendered.
>
> 1. Hoover Factory Service Centers.
>
> 2. Hoover Authorized Warranty Service Dealers (Depots).
>
> [details on servicing omitted]
>
> This warranty does not cover pick up delivery, or house calls; however, if you mail your appliance to a Hoover Factory Service Center for warranty service, transportation will be paid one way.

> While this warranty gives you specific legal rights, you may also have other rights which vary from state to state.

The contrast is striking. For the vacuum cleaner, I got a full, unlimited warranty, and I did not need it. Moreover, I still have a fully functioning vacuum cleaner. For Illustrator, I got a limited warranty, and needed a full warranty, as the limited warranty did not provide a useful remedy. A new copy would behave as the one I had and my money back would leave me with no Illustrator.

The Sharp microwave oven comes with a warranty that says:

> SHARP LIMITED WARRANTY
>
> Consumer Electronics Products
> **Congratulations on your purchase!**
>
> Sharp Electronics of Canada Ltd. (hereinafter called "Sharp") gives the following express warranty to the first consumer purchaser for this Sharp brand product, when shipped in its original container and sold or distributed in Canada by Sharp or by an Authorized Sharp Dealer:
>
> Sharp warrants that this product is free, under normal use and maintenance, from any defects in material and workmanship. If any such defects should be found in this product within the applicable warranty period, Sharp shall, at it's [sic] option, repair or replace the product as specified herein.

The product must perform completely as a microwave oven, and there is no concept of behaving only substantially like a microwave oven. Also, Sharp is guaranteeing that the customer will have a microwave oven somehow, and that he or she will never have to settle for money back and no microwave oven.

> This warranty shall not apply to; [sic]
>
> (a) Any defects caused or repairs required as a result of abusive operation, negligence, accident [sic] improper installation or inappropriate use as outlined in the owner's manual;
>
> (b) Any Sharp product tampered with, modified, adjusted or repaired by any party other than Sharp, Sharp's Authorized Service Centres or Sharp's Authorized Servicing Dealers;
>
> (c) Damage caused or repairs required as a result of the use with items not specified or approved by Sharp, including but not limited to, head cleaning tapes and chemical cleaning agents.
>
> (d) Any replacement of accessories, glassware, consumable or peripheral items required through normal use of the product, such as earphones, remote controls, AC adaptors, batteries, temperature probe, stylus, trays, filters, etc.
>
> (e) Any cosmetic damage to the surface or exterior that has been defaced or caused by normal wear and tear.
>
> (f) Any damage caused by external or environmental conditions such as liquid spillage or power line voltage, etc.
>
> (g) Any product received without appropriate model and serial number identification and/or CSR markings.
>
> (h) Any consumer products used for rental or commercial purposes.

Sharp is careful to exclude causes of damage that are not its fault or are the fault of the customer or normal wear and tear. Notice that nothing for which the customer would really expect Sharp to be responsible has been excluded.

> Should this Sharp product fail to operate during the warranty period, service may be obtained upon delivery of the Sharp product together with proof of purchase to an Authorized Sharp Service Center or an Authorized Sharp Servicing Dealer.
>
> [details on servicing omitted]
>
> This warranty constitutes the entire express warranty granted by Sharp and no other dealer, service center or their agent or employee is authorized to extend, enlarge or transfer this warranty on behalf of Sharp.

The period of the microwave warranty is defined elsewhere in a table.

WARRANTY PERIODS

...	
Microwave Oven	2 years (magnetron 3 additional years part warranty only)
...	

The limited warranty period is a recognition that an appliance decays and that it cannot operate like new forever.

Basically, for appliances, manufacturers warrant that there are no defects, that the appliance behaves as it is specified, and that they will make the appliance run if the customer finds a defect within the warranty period.

It is my belief that if laws were changed forcing software manufacturers to guarantee fitness for purpose or functionality, their procedures would change so that software is released only after the same kind of quality control that appliances are subjected to.

3.2 Liability

This section examines the liabilities borne by the producers of the software products and appliances described in Section 2. Appliance manufacturers are held liable for damages caused by their appliances, e.g., if an appliance blows up, catches fire, etc. Furthermore, if it can be shown that the manufacturer failed to apply accepted quality control procedures for the engineering disciplines involved in the manufacture, the manufacturer can be judged willfully negligent and can be assessed punitive damages. Consequently, an appliance manufacturer applies whatever methods are available for predicting behavior and assuring quality of its products, including testing and modeling. It also arranges for independent verification and validation (IV&V), for example, by the Underwriters' Laboratory, as part of the process of determining the cost of its liability insurance.

The Hoover vacuum cleaner warranty has *no* limitation of liability whatsoever. The Sharp microwave oven warranty has a limitation of liability.

> To the extent the law permits, Sharp disclaims any and all liability for direct or indirect damages or losses or for any incidental, special or consequential damages or loss of profits resulting from a defect in material or workmanship relating to the product, including damages from loss of time or use of this Sharp product. Correction of defects, in the manner and period of time described herein, constitute complete fulfillment of all obligations and responsibilities of Sharp to the purchaser with respect to the product and shall constitute full satisfaction of all claims, whether based on contract, negligence, strict liability or otherwise.

In many places, the law does not permit Sharp to disclaim all liability, particularly of damages or loss caused by a functioning or malfunctioning product. In other words, if a correctly used microwave oven explodes, Sharp is liable for the damages and loss caused by the explosion.

Software developers suffer no such liability. There are few laws specifying their liability. Furthermore, they usually write into their shrinkwrap, mass market licenses a disclaimer for liability for damages beyond the cost of the software itself. Adobe's EULA shouts out a very strong limitation on liability; Microsoft's EULA has a very similar shouted limitation on liability.

> **6. Limitation of Liability**. IN NO EVENT WILL ADOBE OR ITS SUPPLIERS BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, INCIDENTAL, OR SPECIAL DAMAGES, INCLUDING ANY LOST PROFITS OR LOST SAVINGS, EVEN IF ADOBE REPRESENTATIVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY THIRD PARTY. Some states or jurisdictions do not allow the exclusion or limitation of incidental, consequential or special damages, so the above limitation may not apply to you.

In most jurisdictions, the producer has no liability whatsoever for any damages caused by the software's inability to do its function or for any damage done by malfunctioning software. The consumers accept the useless warranty and the limitation of liability. More than that, consumers accept the poor quality software and keep paying for upgrades, which are often little more than corrections of flaws in a product that they already paid for.

It is my belief that if laws were changed forcing liability on software developers, their procedures would change to use all available methods for assuring quality, including inspections, testing, IV&V, and even formal methods. They will do anything to stave off a claim of willful negligence in the event of damages from the execution of their software.

3.3 Mass-Market vs. Bespoke Software and Negotiating Power

Note again that I am talking about consumer software developed at a producer's own expense and risk for the mass market. For bespoke software, especially for systems with high reliability and safety concerns such as in aircraft, automobiles, telecommunications, and process control, the producer warrants the product and is subject to liability as a result of the contract negotiated face to face between the client and producer. Here, the client has the power to force the producer to warrant the product and accept liability, because the client can always go to another producer. In the consumer market, in which there is no face-to-face negotiation of a contract, a contract warranting nothing and limiting the producer's liability is foisted on the consumer through the shrink-wrap mechanism. Because for a given function, there is often only one product that runs on a customer's system or that all those interacting with the customer can use, the customer is forced to accept this product and its license; the producers have the power to force consumers to accept an agreement that strongly favors the producers. Perhaps this imbalance of power is the reason that consumers accept the poor quality of software and the unfavorable terms of the shrinkwrap consumer software license agreement.

3.4 Wrap Up

This discussion of warranty and liability is best concluded by examining a letter written by Justus Pendleton of Somerville, MA to *IEEE Computer*, January 1999 [28]. The letter was written ostensibly to to point out that it is no easier to verify fitness for use of normal commercial shrinkwrapped software than of open-source software. He was replying to an earlier letter to the same journal that claimed that open-source software offers full protection only if the user has the resources to conduct a full inspection. Pendleton wrote:

> There is a fitness-for-use disclaimer in virtually all software that usually says something to the effect ''this [information, computer program] is being provided with all faults, and the entire risk as to satisfactory quality, performance, accuracy, and effort is with the user.'' The buyer of shrinkware as to either take the vendor's word that the software is fit to use or subject it to black-box testing (the results of which cannot be published without the vendor's explicit and prior permission), which is arguably more difficult that a thorough inspection of source code. ... These are the same vendors that tell us the next version, which is due out next month sometime, will fix all the problems we are having.

4 Quality Assurance Methods and Warranties and Liabilities

In a number of engineering disciplines, there are systematic and sometimes formal procedures for verification and validation that are to be followed while the product is in design stage. Electrical engineers routinely apply mathematical models of electronics to determine if their designs will function correctly and will meet safety requirements. Civil engineers and architects routinely apply mathematical models of structures to verify that the structures they are designing will support the load to which they will be subjected and that they will withstand the environmental forces that may push on them. The reason that these engineers routinely apply their QA procedures is that if they do not and the product does not work as it is supposed to, their employers may be inundated by customer complaints, may suffer massive returns with refunds, and may, in the worst case, be sued for damages. The employers may then take disciplinary and, in some cases, job action against the engineers responsible for the malfunctioning product. Also, if these engineers do not apply their QA procedures and the product causes damages, the failure to apply the QA procedures in the construction of the product may subject the manufacturer to a negligence claim and punitive damages beyond the just the base cost of the damages.

In these engineering disciplines, the manufacturers establish procedures to be followed during design, development, and manufacturing. These procedures include a variety of tests, ranging from inspection of documents, through actual usage of prototypes of and samples of the developed products, to exercising mathematical models. The manufacturers require employees to follow these procedures and to document that they have followed the procedures. The documentation may be subpoenaed in a damages lawsuit. Failure to follow these procedures subjects the offending employee to disciplinary action and, in some cases, job termination. These procedures and penalties for failure to follow the procedures is the manufacturer's best defense against a negligence claim.

The professional requirements for a medical doctor or physician are instructive. A physician is held to *the standard of care* in his or her community. Failure to provide at least the current standard of care may subject the physician to a negligence complaint and to malpractice action. The definition of this standard varies and depends on

1. what is taught at medical school,
2. the results of recent medical research, and
3. what the physicians in the community regularly do, given the resources available.

The community standard of care is determined case-by-case in malpractice cases from the testimony of expert witnesses, usually other physicians*.

In medicine, the standard of care for a community is a baseline and may not be all that close to the state of the medical art. It consists of what the doctors in the community consider to have been demonstrated as effective treatment, modulo the facilities and resources available to carry it out. While it is not required for a physician to apply the latest treatments, which may be only experimental, it is not an acceptable defense in a malpractice suit to say that the applied out-of-date treatment is what the physician learned in medical school. The physician is required to keep up to date and learn new treatments that have been demonstrated effective against diseases in his or her specialty. The standard of care for a community evolves continually with new treatments established by research as effective.

Anyone with a duty to be careful in a treatment is considered negligent and is liable for damages if he or she has not applied the accepted standard of care, the care causes damages, and there was no independent, intervening cause of the damages. The standard of care is higher for a relevant professional than for others. For a non-physician, the standard of care for medical treatment is what the reasonable person-in-the-street would do in the circumstances. For the professional physician, not to apply the community's standard of care for

* These expert witness doctors are paid by one side or another in the case. Thus, it is not hard to find doctors who, for a fee, will testify that another doctor's care was not up to the standard of care for the community. It is also not hard to find other doctors who, for a fee, will testify to the exact opposite. In the end the judge or the jury have to decide if the care was up to the standard.

physicians is considered malpractice.

In medicine, the standard of care does not require using not-yet-widely used treatments and, in fact, may require *not* using them, especially if they are as yet unproved. However, in other areas, one might be expected to use a new technology even it is not yet widely used. In such a case, the standard of care drives adoption of new techniques. There was a famous case from the 1920s or 1930s in which the operators of a tugboat, the *T. J. Hooper*, were held liable for the boat's sinking in a storm because there was no radio on board with which to listen to weather reports. The operators were held liable even though, at the time, most boats did not have radios. This case spurred the adoption of radios as standard equipment on board boats.

5 Loss of Exclusion of Warranty and Liability

What will happen if warranty and liability limitations for software producers are brought in line with those of other manufactured products, and software producers become as accountable for product quality as other manufacturers. Please allow me to speculate. I believe that the software producers will have to start applying community and professionally accepted standards of care, both to produce more reliable software and to serve as a defense against liability should products cause damages despite the care. They will need to establish procedures that must be followed during specification, design, development, and deployment. These procedures will include a variety of tests, ranging from inspection of documents, through uses of prototypes and production code in runs against test data, to formal model checking and verification. They will require the software engineers working for them to follow these procedures and to document that they have done so. Finally, they will provide for disciplinary action and even job termination for failure to follow these procedures.

Similar procedures are established by many software producers in order to obtain CMM or ISO 9000 certification [27]. Because of the artificial imposition of the procedures that have no observable direct positive impact and seem to mire a project in process, many employees doubt the effectiveness of the procedures and may even subvert their imposition. Moreover, many software manufacturers, with no chance for or interest in government contracts, simply do not bother with certification at all. Loss of warranty and liability limitation will force all software producers to adopt systematic QA procedures, possibly those suggested by CMM or ISO. In addition, the effect of failure to follow the procedures will be felt more directly and swiftly, in the form of job and legal actions, thus encouraging better compliance by employees.

Indeed, here we may have the solution to the inexorable pressure of the rush to market that give a high incentive to premature release of software. The loss of warranty and liability limitation changes the economics of early release. Early release may increase exposure to warranty and liability claims, which can be very costly. Thus, there would be a higher incentive to slowing down to release higher quality SW. It would become a tradeoff of exposure to warranty and liability claims versus loss of market.

These procedures will become the accepted standard of care against which all work will be compared, particularly if the work has led to a substandard product or one which has caused damages. Of course, in software development, the standard of care will vary depending on the product. The more critical the product, the higher the standard of care for its development. For software driving a system on which lives depend, the standard of care will be considerably higher than for software driving a recreational game. For some life-critical systems, the standard of care would likely include formal methods such as model checking and possibly even formal verification. For a program to play solitaire, the standard of care would be considerably lower, probably including only inspection and basic testing.

The standard of care for software development will depend on

1. what is taught in software engineering degree programs,
2. the results of recent software engineering research, and
3. what software engineers in the community regularly do, given the resources available and the domain of the software.

Quite likely software engineering will be judged to be a field in which new techniques, not yet widely adopted, will be expected to be used when the situation warrants it.

Where are formal methods in all of this? Formal methods are taught in software engineering degree programs, formal methods are explored in software engineering research, formal methods are used in the most critical projects. Finally, formal methods, while not widely adopted, have been shown to be of benefit in development of critical software. Therefore, it seems clear that formal methods will be part of the standard of care for some software developments and that the exposure of software producers to liability will drive them to adopt formal methods for the development of critical software, and possibly, of some less than critical software.

6 The Software Engineering Profession

There is a move to make software engineering a full-fledged engineering profession [24,25]. Just as the practitioners of other professions, engineering, medicine, or others, are expected to apply the profession's standard of care in their work or face malpractice action, so will the software engineer be expected to apply software engineering's standard of care. After all, an engineer's responsibilities include making sure products he or she produces are fit for use, contrary to what current software warranties claim. If this standard of care includes formal methods, software engineers would be compelled to apply formal methods in appropriate circumstances. The software engineer who does not apply this standard of care would find himself out of a job or facing legal malpractice action.

Normally, the company that produces a product is liable for the product and the individual employees are not. However, an individual licensed engineer assumes liability for those products whose fitness he or she has guaranteed with his or her signature. This liability borne by the licensed engineer who signs off on a product is scary and probably accounts for the resistance of many current software developers, who call themselves "software engineers", to licensing of software engineers under standard engineering charters. The engineer can lose his or her profession and face severe legal consequences if a product he or she develops and guarantees fails.

7 Conclusion

Once product warranty and liability applies to software products, a producer of software will be compelled not to release software until it can guarantee that it behaves as it is supposed to, for fear of consequences such as a flood or complaints, having to refund lots of buyers, having to recall the product, having to stand behind a faulty product, and possibly even paying damages if the product causes damage as a result of not behaving as it is supposed to. I believe that to meet the required level of quality, software producers will be force to establish systematic QA procedures. They will be forced to put teeth into the procedures in order to force employees, the software engineers, to comply with these procedures. The software engineers will face disciplinary action and possible termination for failure to follow the procedures. The procedures will likely include formal methods for certain classes of critical systems.

Perhaps one day, commercially available software will be as reliable as commercially available appliances. While appliances are by no means pefect, they are a whole lot more reliable than software. I could live with software being as good as my microwave oven!

8 Added in Proof

Just as I was preparing this paper for submission, an extremely relevant article authored by Joseph Menn, a *Times* staff writer, appeared in the newspapers on 4 February 2000. It also appeared in the WWW at

http://www.latimes.com/
business/updates/lat_rights000204.htm.

The article at the web site is titled "Software Makers Aim to Dilute Consumer Rights" with a subtitle of "Technology: Companies push legislation at state level that would dramatically alter contract law in their favor."

> Microsoft Corp. and other powerful software companies are quietly pushing state

> legislation across the nation that would dramatically reduce consumer rights for individuals and businesses who buy or lease software and database information.
>
> The push comes as software companies are beefing up their lobbying effort to pass favorable laws while their industry is at peak popularity among politicians who want to keep their local economies booming, consumer groups say.
>
> "[This] is an example of newly powerful software giants using the promise of high-tech jobs to push through legislation that restricts consumer and business-customer rights," said James Tierney, former Maine attorney general, who opposes the effort.
>
> The tech bills spring from a proposal with an arcane name, the Uniform Computer Information Transactions Act (UCITA). Should states pass this legislation, the impact on consumers would be dramatic:
>
> ....
>
> But in dozens of ways, large and small, the bills tip the balance of power toward software companies, according to law professors, consumer groups, more than 20 state attorneys general and some corporate software buyers that are beginning to organize an opposition to the UCITA campaign.
>
> If these UCITA-sponsored bills pass, "it will dramatically change the law," said Herschel Elkins, head of the California attorney general's consumer department. He said the legislation would put buyers into a legal corner with little way out. "It's pay first, find out what you bought later," he said. "The refund right disappears when you click twice on 'I agree.' "
>
> ...
>
> "It's very difficult to understand," [Temple University law professor Amy] Boss said of the bill. Under the legislation, customers who install software in their computers have already lost some of their basic rights, she said. The tech bill "gives the consumer no way to disagree with the terms," she said.
>
> Microsoft's [Rick] Miller declined to discuss some of the complex bill's provisions. Other supporters of the legislation said its critics misunderstand the effect of the measure.

It can be even worse than I thought. In Section 3.3, I observed that the public seems to accept software producers' claims of no warranty and no liability. However, it is not clear that the courts would accept these claims if enough members of the public were to sue. The courts would likely apply standards for normal consumer products, if for no other reason than judges and juries really do not understand software. UCITA, however, would override any such court decision by explicitly legislating the provisions of no warranty and no liability found in most shrink-wrapped licenses.

Acknowledgments

I thank Connie Heitmeyer and Dino Mandrioli for bibliographical references and comments and suggestions that led to improvements in the paper. I thank David Kay for an e-mail discussion about standards of care for medical doctors. Finally, I thank Egon Boerger, Martin Feather, and three anonymous reviewers for their sharp criticisms of a previous version of this paper. These criticisms led to a major revision of the paper. I was supported in parts by a University of Waterloo Startup Grant and by NSERC grant NSERC-RGPIN227055-00.

References

[1] *IEEE Computer* **23**(9) (September 1990), Special Issue.

[2] *IEEE Software* 7(5) (September 1990), Special Issue.

[3] *Proceedings of the Workshop on Industrial-Strength Formal Specification Techniques,* IEEE Computer Society, Boca Raton (April 1995).

[4] *Proceedings of the Fourth NASA Langley Formal Methods Workshop,* NASA Conference Publication 3356, Hampton, Virginia (September 1997).

[5] *Journal Systems and Software* **40**(3) (March 1998), Special Issue.

[6] Berry, D.M., "Formal Methods, the Very Idea, Some Thoughts on Why They Work When They Work," *Electronic Notes in Theoretical Computer Science* **25**, Elsevier (1999), http://www.elsevier.nl/locate/entcs/volume25.html.

[7] Bharadwaj, R. and Heitmeyer, C., "Applying the SCR Requirements Method to a Simple Autopilot," in *Proceedings of the Fourth NASA Langley Formal Methods Workshop,* NASA Conference Publication 3356, Hampton, Virginia (September 1997).

[8] Bowen, J.P. and Hinchey, M.G., "Seven More Myths of Formal Methods," Technical Report PRG-TR-7-94, Oxford University Computing Laboratory (1994).

[9] Bowen, J.P. and Hinchey, M.G., "Ten Commandments of Formal Methods," Technical Report, Oxford University Computing Laboratory and University of Cambridge Computer Laboratory (1995).

[10] Chan, W., Anderson, R., Beame, P., Burns, S., Modugno, F., Notkin, D., and Reese, J.D., "Model Checking Large Software Specifications," Technical Report, Computer Science Department, University of Washington, Seattle, WA (September 1997).

[11] Ciapessoni, E., Coen-Porsini, A., Crivelli, E., Mandrioli, D., Mirandola, P., and Morzenti, A., "From Formal Models to Formally-Based Methods: An Industrial Experience," *ACM Transactions on Software Engineering and Methodologies* **8**(1), p.79–113 (January 1999).

[12] Cleland, G. and MacKenzie, D., "Inhibiting Factors, Market Structure and the Industrial Uptake of Formal Methods," pp. 46–60 in *Proceedings of the Workshop on Industrial-Strength Formal Specification Techniques,* IEEE Computer Society, Boca Raton, FL (April 1995).

[13] Easterbrook, S. and Callahan, J., "Formal Methods for Verification and Validation of Partial Specifications: A Case Study," *Journal Systems and Software* **40**(3), p.199–210 (March 1998).

[14] Gerhart, S.L., "Program Verification in the 1980s: Problems, Perspectives, and Opportunities," ISI/RR-78-71, USC Information Sciences Institute, Marina Del Rey, CA (August 1978).

[15] Gerhart, S.L., "Applications of Formal Methods," *IEEE Software* 7(5), p.6–10 (September 1990).

[16] Gilb, T. and Graham, D., *Software Inspection,* Addison Wesley, Wokingham, UK (1993).

[17] Hall, A., "Seven Myths of Formal Methods," *IEEE Software* 7(5), p.11–19 (September 1990).

[18] Heimdahl, M.P.E. and Leveson, N.G., "Completeness and Consistency in Heirarchical State-Based Requirements," *IEEE Transactions on Software Engineering* **SE-22**(6), p.363–377 (June 1996).

[19] Hinchey, M.G. and Liu, S., *Proceedings of First International Conference on Formal Engineering Methods,* IEEE Computer Society, Hiroshima, Japan (November 1997).

[20] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming," *Communications of the ACM* **12**(10), p.576–580,585 (October 1969).

[21] Jackson, M., "Formal Methods and Traditional Engineering," *Journal of Systems and Software* **40**(3), p.191–194 (March 1998).

[22] Jones, C.B., "Whither Formal Methods: A Plea to Investigate New Applications," pp. 5 in *Proceedings of First International Conference on Formal Engineering Methods,* ed. M.G. Hinchey and S. Liu, IEEE Computer Society, Hiroshima, Japan (November 1997), Invited Lecture.

[23] Knight, J.C., DeJong, C.L., Gibble, M.S., and Nakano, L.G., "Why Are Formal Methods Not Used More Widely," pp. 1–12 in *Fourth NASA Langley Formal Methods Workshop,* NASA Conference Publication 3356, Hampton, Virginia (September 1997).

[24] McConnell, S. and Tripp, L., "Professional Software Engineering: Fact or Fiction?," *IEEE Software* **16**(6) (November/December 1999).

[25] Parnas, D.L., "Software Engineering: An Unconsummated Marriage," *Communications of the ACM* **40**(9), p.128 (September 1997).

[26] Parnas, D.L., ""Formal Methods" Technology Transfer Will Fail," *Journal Systems and Software* **40**(3), p.195–198 (March 1998).

[27] Paulk, M.C., Curtis, B., Chrissis, M.B., and Weber, C.V., "Key Practices of the Capability Maturity Model," Technical Report, CMU/SEI-93-TR-25, Software Engineering Institute (February 1993).

[28] Pendleton, J., "Shrinkwrap Is No Safer," *IEEE Computer* **32**(1), p.9 (January 1999).

[29] Pressman, R., "Software According to Niccolò Machiavelli," *IEEE Software* **12**(1), p.101–102 (January 1995).

[30] Wing, J.M., "A Specifier's Introduction to Formal Methods," *IEEE Computer* **23**(9), p.8–24 (September 1990).

[31] Woodcock, J.C.P. and Larson, P.G., *FME'93 Industrial-Strength Formal Methods,* Springer, LNCS 670 (1993).

Static Analysis for Program Generation Templates[1]

Valdis Berzins
Naval Postgraduate School
Monterey CA 93943 USA

Abstract

This paper presents an approach to achieving reliable cost-effective software via automatic program generation patterns. The main idea is to certify the patterns once, to establish a reliability property for all of the programs that could possibly be generated from the patterns. We focus here on properties that can be checked via computable static analysis. Examples of methods to assure syntactic correctness and exception closure of the generated code are presented. Exception closure means that a software module cannot raise any exceptions other than those declared in its interface.

1. Introduction

Our goal is to provide cost effective means for creating reliable software. We are addressing the issue by improving the technology for automatic software generation, with particular attention to reliability issues.

We take a domain specific view of this process: a domain is a family of related problems addressing a common set of issues. A domain analysis identifies the problem and issues, formulates a model of these, and determines a corresponding set of solution methods. Users of the proposed computer-aided software generation system describe their particular problem using a domain specific problem modeling language that provides concrete representations of problems in the domain. The system then automatically determines which solution methods are applicable, customizes them to the specific problem instance described using the modeling language, and then automatically generates a program that will solve the specified problem.

We seek to provide tool support for the above process that can be applied to many different problem domains, and that can generate code in any programming language. Therefore we seek uniform and effective methods for generating software generators of the type described above, given definitions of the problem modeling language, the target programming language, and the roles for synthesizing solution programs. A simple architecture for this process is shown in Figure 1.

The specific goals of this paper are: (1) to provide a simple example of a language for expressing software patterns that are specific enough to be used as synthesis rules and (2) to provide examples of static rules in this language. We address the problems of certifying that all programs which can be generated from a given set of rules: (1) are syntactically correct and (2) will not raise any exceptions other than those explicitly specified in an interface description.

This is a step towards a coordinated system of static and dynamic checks, to be performed on program synthesis rules. Our hypothesis is that the most cost effective way to improve software quality is to systematically improve and certify the rules used to generate a domain-specific software generator. This approach directly addresses the issue of correctly implementing given software requirements. It also indirectly addresses the issue of getting the right requirements, because it should eventually enable rapid prototyping of product quality systems by problem domain experts, who need not be software experts. If the requirements are found to be inappropriate, the domain experts will simply update the problem models and regenerate a new version of the solution software.

We will refer to the software generation patterns as templates. Our rationale for the claim of cost effectiveness is that the benefits of quality improvements to the templates can be extended to all past and future applications of the generators - by regenerating the generator using the improved templates and then regenerating the past applications. The regeneration process can be completely automated, thereby reducing labor costs, eliminating a source of random human errors, and speeding up the process of repairing a known fault throughout a large family of software systems.

[1] This research was supported in part by the U. S. Army Research Office under contract/grant number 35037-MA and 40473-MA, and in part by DARPA under contract #99-F759.

The relation to the theme of this workshop is that fast moving scenarios can be addressed by automatically generating new variants of the software that reflect changing issues in the problem domain. Our approach should reduce the explicit quality assurance efforts needed each time the software is changed. By amortizing the quality assurance effort applied to the template over many applications of the same templates, we can reduce quality assurance costs. The benefits increase with the number of systems generated from the same templates.

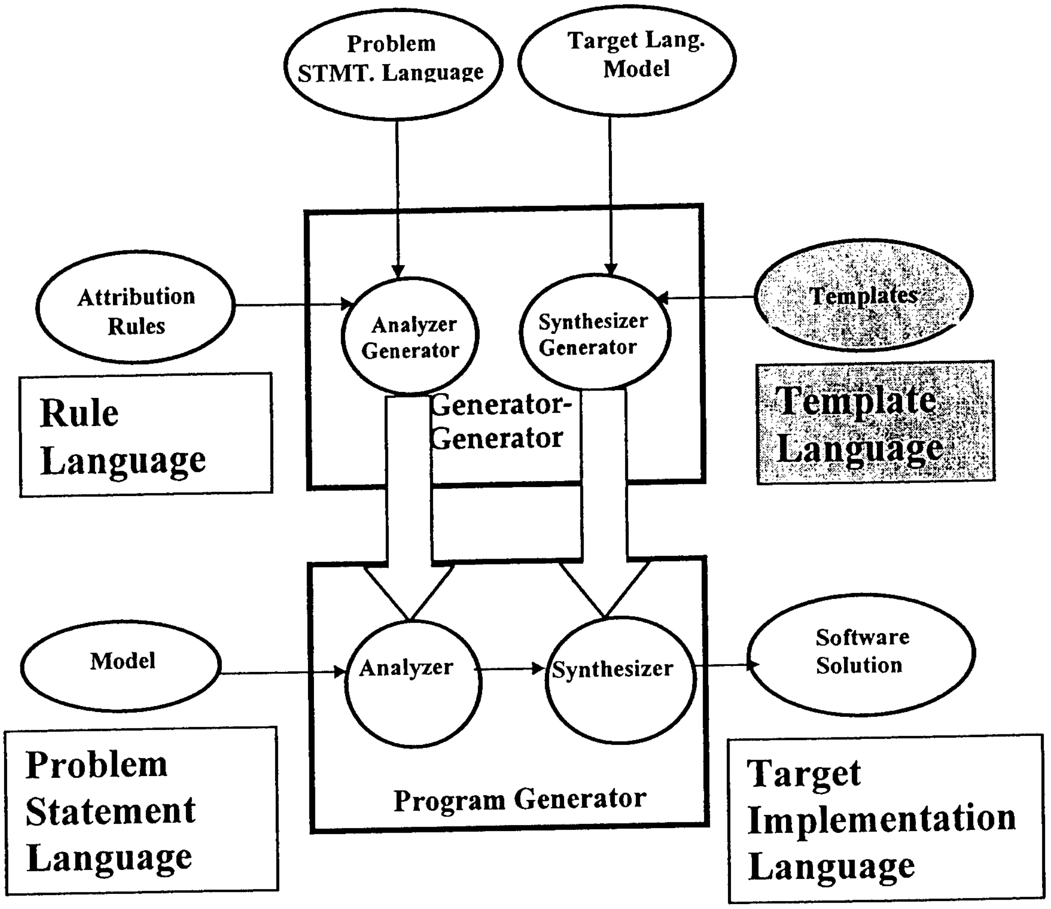


Figure 1. Model-Based Software Generator Architecture

This paper focuses on static checks that can be completely automated. Our research is also addressing testing and debugging of program synthesis rules and proofs of rule properties that require human assistance with deeper reasoning. These efforts are outside the scope of the current paper, which is organized as follows:

- Section 2 formalizes software generation patterns and defines a uniform construction to obtain a template language for any target programming language.
- Section 3 describes methods for statically certifying syntactic correctness generated code, and gives an example.
- Section 4 does the same for analysis of exceptions.
- Section 5 contains comparisons to previous work
- Section 6 presents conclusions.

2. Template Languages

The purpose of a template language is to define software synthesis patterns for a given target language. We create such languages based on a functional object model of code generation templates. We take a functional (i.e. side-effect-free) approach because this simplifies the algebraic basis of the approach and supports effective static analysis methods such as those presented in Section 3 and 4.

We view template languages as extensions of the corresponding target programming languages. Because many different programming languages are created, we will need many different template languages. However, all of these can be defined at once by providing uniform construction such as that shown in Figure 2.

This is a very simple construction, but it is very expressive. In addition to providing substitution of actual values for generic parameters, as in the generic units of Ada and the templates of C++, our construction includes conditionals that are evaluated at code generation time, and the ability to invoke other templates. Recursion is included.

Template_language = {template, formal_def, template_expression}

DEF_TEMPLATE(id[template], type, seq[formal_def], template_expression):
    template -- where type Î ∈ target_language

DEF_FORMAL(template_parameter, type): formal_def
    -- declares the type of a formal parameter

template_parameter < {id[any], template_expression}
IF(template_expression, template_expression, template_expression):
    template_expression
APPLY(id[template], seq[template_expression]): template_expression

template_expression < target_language

Figure 2. Template Abstract Syntax

The construction depends heavily on the use of inheritance in object-oriented modeling of programming languages. The situation is illustrated in Figure 3.

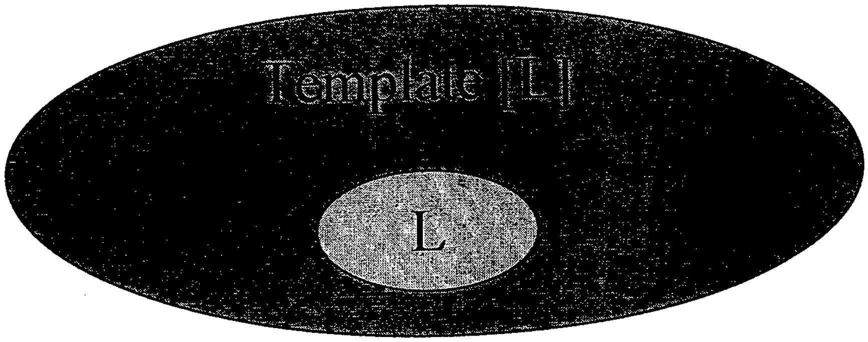


Figure 3. Generic Template Language

In object-oriented modeling, class-wide types[2] are viewed as open and extensible. Specifically, each time we add a subclass with a new constructor, we add more instances to the class-wide type, thus extending its value set.

We model the abstract syntax of a language using a type for each kind of semantic entity. In a properly constructed abstract syntax, there should be one such type for each non-terminal symbol. Each constructor of these types corresponds to a production of the grammar. Subclass relationships, denoted by "≤", specify that every instance of the subclass is also an instance of the parent class. Multiple inheritance is allowed. For example, in line 6 of Figure 2 says that every template parameter is a kind of identifier, and also is a kind of template expression. This kind of subclass relationship is used to incorporate reusable types in a library of programming language building blocks, such as identifiers, and to specialize reusable concepts to the application, such as template expression. If T is a type and S is a set of types, T<S means T is a subclass of each element of S. This represents multiple inheritance.

[2] This is Ada 95 terminology. The instances of a class wide type include its direct instances and those of all its subclasses, transitively.

Subclassing is also used to interface between a target programming language and its extensions. In Figure 2, "target-language" denotes the set of types comprising the abstract syntax of the target language. Figure 4 shows a very simple example of a target language that illustrates how this works.

```
target_language = (stmt, exp)

assign(var, exp): stmt
if(exp, stmt, stmt): stmt

integer < exp   -- integer literals
var < {id[any], exp} -- program variables
apply(id[function], seq[exp]): exp -- operations

subtype rule:  x < y ==> id[x] < id[y] where x, y ∈  type
```

Figure 4. Example: Micro Target Language

The example in Figure 5 defines a code generation pattern that embodies Newton's method for polynomial evaluation, which is optimal in terms of number of evaluation steps needed. This is a very simple example of a code generation pattern that is nevertheless realistic, because it embodies a solution method. The example also illustrates the use of all the constructs in the template language. We use infix syntax for the exp constructors * and + to improve legibility (e.g. x*y is short for the term apply(*, x, y)).

An additional benefit of considering the abstract syntax to be an algebra rather than a tree is that we can used well-studied transformation rules. In particular we can associate equational axioms with the programming language types that define normal forms. Figure 5 illustrates the use of such axioms as rewrite rules that simplify the code produced by the generator in a follow-on normalization process. This is one way to incorporate optimizations into the program generation process, which is useful for unconditional transformations.

```
TEMPLATE evaluate_polynomial (v: var, c: seq[integer]): exp
    -- c contains coefficients of a polynomial, lowest degree first
    IF not (is_empty (c) )  -- use operations of boolean and seq
    THEN v * (evaluate_polynomial (v, rest(c))) + first (c)
    ELSE 0
END TEMPLATE

Template application evaluate-polynomial(x, [1, 2, 3]) generates
    x * (x * (x * 0 + 3) + 2) + 1

Normalization with integer rules i * 0 = 0, i + 0 = i reduces to
    x * (x * 3 + 2) + 1
```

Figure 5. Example: Generation Pattern

Code generation using the template language is a very much like evaluation in a functional programming language with call-by-value semantics. Analysis of templates can take advantage of equational reasoning, substitution, and structural induction. The limitation to primitive recursion facilitates the latter. The recursion in the example is structural because **rest** is a partial inverse for the sequence constructor **add** (i.e. rest(add(x, s)) = s).

3. Syntactic Correctness of Generated Code

We treat the abstract syntax structures of the target language as the values of the abstract data types representing the programming language. We require these types to provide a pretty printing operation that outputs such objects as text strings according to the concrete syntax of the target language, with a readable format. Establishing correctness of these pretty printing operations is straightforward, and in fact their implementations can be generated from an appropriately annotated grammar for the concrete syntax.

Given trusted pretty printing operations for the object model of the target language, syntactic correctness of the output reduces to the type-correctness of the ground terms generated by the evaluation of the templates. This can be checked using a simple type system for the template language and conventional type checking methods. Note that we are referring to the types associated with the signatures of the constructors in the object model of the target programming language, rather than the types within the target programming language, which may not even be a typed language. The process is illustrated Figure 6. The computed type annotations are shown in *italics*. The type annotations associated with the implicit induction step, where the type signature of the template itself is used, is highlighted in ***bold italics***. The indentations of the type annotations reflect the structure of the derivation.

```
TEMPLATE evaluate_polynomial (v: var, c: seq[integer]): exp
   IF not (is_empty (c : seq[integer]) : boolean ) : boolean
   THEN  +(  *( v                                          : var,
                 evaluate_polynomial
                     (v                  : var,
                       rest(c: seq[integer] ) : seq[integer]) : exp
                                                           ) : exp
                 first (c: seq[integer] )                    : integer
                                                           ) : exp
            --term form of v* evaluate_polynomial (v, rest(c)) + first (c)
    ELSE 0                                                    : integer
END TEMPLATE

Types conform because integer < exp and var < exp

Relevant signatures: +(exp, exp) :exp, *(exp, exp) :exp,
                     first(seq[T]): T, rest(seq[T]): seq[T],
                     is_empty(seq[T]): boolean, not(boolean): boolean
```

Figure 6. Example: Syntactic Correctness of Generated Code

Note that induction has been carried out implicitly, as a routine step of the type checking calculation. This is sufficient to establish partial type correctness of the templates, which implies syntactic correctness of all code that could be generated by the template, it does not automatically guarantee total correctness, because we still have the possibility that evaluation of the template might fail to terminate.

Total correctness is established by the type check if we check that all recursions are primitive. The example satisfies this condition because rest is a partial inverse of the compound sequence constructor; rest(add(x,s)) = s. This means that the induction is in fact structural, and hence that evaluate_polynomial is total. Thus the template will produce syntactically correct code for all input values that conform to the type signature of evaluate_polynomial.

We note that given declarations of the target language constructors that define the abstract syntax and the corresponding partial inverse operations, it is straightforward to automatically check that all recursive calls are primitive with respect to any given parameter position. This implies that structural induction can be applied uniformly and completely automatically in this context. Furthermore, our experience suggests that structural recursions are sufficient to define the code generation templates needed in practice, and that template designers can live within the restriction to structural recursions without undue hardships.

4. Exception Closures for Generated Code

One common source of software failure is unhandled exceptions. This section explains a method for certifying that all programs generated from a given template cannot generate any unhandled exceptions when placed in a context that handles a specified set of exceptions.

Our approach is to refine the type system to record the set of exceptions that might be raised by the evaluation of any expression of the target language. A similar structure can be used to analyze the set of exceptions that might be raised by execution of a statement of the target language.

The refinement replaces the single target language type exp with a parameterized family of types exp[set[exception]]. The intended interpretation of this type structure is that evaluation of an expression of type exp[S] might raise an exception e only if $e \in S$. Since we do not require all exceptions in S to be producible, this family of types has a rich subclass structure defined by the following relation:

$$S1 \subseteq S2 \Rightarrow exp[S1] \leq exp[S2]$$

The type signatures of an operation are specified explicitly for argument expression type that cannot raise any exceptions, and are extended to all other types by the following rule, which describes the essential pattern for propagating exceptions:

$$F(exp[\emptyset]) : exp[S1] \Rightarrow f(exp[S2]): exp[S1 \cup S2]$$

The rule for operations with multiple arguments is similar. Similar rules apply to language constructs representing exception handlers. Exception handlers follow rules of the form

$$(TRY\ exp[S1]\ CATCH\ e\ USE\ exp[S2]): exp[(S1\text{-}\{e\}) \cup S2].$$

Figure 7 shows the exception analysis for our running example. The parts added to the version in Figure 6 are underlined.

```
TEMPLATE evaluate_polynomial (v: var, c: seq[integer]): exp [{ovfl}]
 IF not (is_empty (c: seq[integer] ): boolean ): boolean
    THEN +(*(v: var,
              evaluate_polynomial(v: var,
                 rest(c: seq[integer] ): seq[integer] ): exp [{ovfl}],
              first (c: seq[integer] ) : integer ): exp [{ovfl}]
     -- term form of  v * evaluate_polynomial (v, rest(c)) + first (c)
    ELSE 0: integer
END TEMPLATE
```

Types conform because integer < exp[∅] ≤ exp[{ovfl}] and
var < exp[∅] ≤ exp[{ovfl}]

Relevant signatures: +(exp, exp): exp [{ovfl}], *(exp, exp): exp [{ovfl}], first(seq[T]): T, rest(seq[T]): seq[T], is_empty(seq[T]): boolean, , not(boolean): boolean

Figure 7. Exception Closure of Generated Code

Note that we require the author of the template to specify in the type declaration of a template the set of exceptions the generated expression is allowed to raise. This acts as an induction hypothesis in our exception analysis, which is used when analyzing the recursive call of evaluate-polynomial. It also provides useful information for the user of the generated code.

The analysis shown in the figure establishes a partial exception closure: it guarantees that all expressions generated by the template can at most raise only the exception **ovfl** representing integer overflow.

To establish a total exception closure, we have to address clean termination of the template expansion at program generation time. The primitive recursion check explained in the previous section guarantees there will be no infinite recursions, so that termination is guaranteed. However, for clean termination, we must also check that evaluation of the template will not raise any exceptions at program generation time.

Note that the analysis in Figure 7 addresses run-time exceptions. When viewed as constructors of the abstract syntax, + and * are total operations. Overflow exceptions can occur only when those expressions are evaluated, not when they are constructed.

The sequence operators **first** and **rest** are different: they are partial query methods of the abstract syntax, not total constructors. If applied to an empty sequence, they raise a sequence underflow exception. However, this can occur only at program generation time, not at run time.

To certify clean termination of template at program generation time requires a type refinement to record sets of possible exceptions and an additional kind of type refinement to record domains of partial methods such as **first** and **rest**. We can introduce a subtype nseq[T, S] < seq[T, S] consisting of the nonempty sequences, and refine the signatures of the partial sequence operations first and rest as follows.

first(nseq[T, ∅]): T[∅], rest(nseq[T, ∅]): seq[T, ∅]
first(seq[T, ∅]): T[seq_underflow], rest(seq[T, ∅]): seq[T, {seq_underflow}]

Type analysis requires a bit of inference in this case, because we have to use the guard of the template language conditional IF together with the rule

s : seq[T, S] and not is-empty (s) ⇒ s: nseq[T, S]

This inference is easy because the guard matches the subtype restriction predicate for nseq[T].

This match did not occur by accident - the purpose of the guard is precisely to ensure that the operations first and rest are used only within their domain of definition. In the interests of being able to produce certifiably robust code, we claim that it would not be unduly burdensome to require that template designers associate domain predicates with all partial operations, and use those domain predicates explicitly in guards whenever they are needed to ensure the partial operators are used within their proper domains of definition. For example, **first** could be associated with a domain predicate

first-ok (seq[T]) : boolean where
first-ok (s) = not (is-empty (s)).

This would enable a fast and shallow analysis of guard conditions to certify absence of exceptions in cases like this. Some such restriction is necessary for practical engineering support because the problem of checking whether an unconstrained guard condition implies the domain predicates of arbitrary guarded partial operations is undecidable.

An alternative is an exception analysis that includes exceptions in the closure even in cases where the guard condition ensures they will never arise. We suggest that it is more practical to handle a common subset of efficiently recognizable forms, and to ask designers to work within the constraints of those recognizable forms. We believe this would be less burdensome than the alternative of manually analyzing the cases where a type check insensitive to guard conditions would nominate exceptions that cannot in fact occur, and that it would lead to a more robust software by making it practical to do complete analysis of exception closures. For example, we could require the example of Figure 7 to be written in a stylized form that looks like the following:

IF first-ok (c) and rest-ok (c)
THEN ... **first** (c) ... **rest** (c) ...

A similar type check would have to be applied to the implementations of **first** and **rest** to ensure that they would in fact terminate cleanly whenever the domain predicates are true.

5. Comparisons to Previous Work

One of our contributions has been to formalize and abstract the idea of a program generation pattern, to make it independent of the details of the target programming language and the process of instantiating the patterns. The purpose of this was to create context in which systematic analysis of program generation patterns becomes possible and in some cases becomes decidable.

Program generation patterns have been evolving for a long time. Macros are an early form of the idea. However, macros are notoriously difficult to analyze, partially because they traditionally operate on uninterpreted text. This makes the connection between macro definitions and the behavior they ultimately denote complicated and potentially very indirect. The macros in LISP are an improvement because they are based on abstract syntax trees rather than characters. However, in this context a second source of complexity becomes apparent: a macro can expand to produce another macro, and the number of expansion steps before the generated source code actually appears is potentially unbounded. This makes the system very difficult to analyze. At the other extreme are the generic units of Ada. These are strongly typed, clearly connected to the abstract syntax of the language, and the results of instantiating them are easy to analyze. However, they do not allow conditional decisions at instantiation time, and are restricted in the sense that the abstract syntax trees of all possible instantiations have exactly the same shape, up to substitution for the formal parameters of the pattern. A language-independent version of the idea can be found in [5], although this appears to be largely text-based.

Another aspect of our approach is to model languages as algebras rather than as abstract syntax trees. A hint of this idea appears in [4], although it is not exploited there for enabling analysis to any significant degree. The work of the CIP group [1] develops this idea further and takes advantage of the reasoning structures that come with the algebraic modeling approach, such as term rewriting and generation induction principles. This suggests extension to a full object-oriented view, which includes inheritance. The Refine system is the earliest context we know of where grammars are treated as object models with potential inheritance structures, although the documentation does not give any hint about the significance of this capability. In this paper we demonstrate the usefulness of algebraic models of syntax with inheritance, for defining language extension transformations that can be applied to all possible target languages.

Another theme is lightweight inference [2]. We have demonstrated that some useful types of static analysis for program generation patterns can be performed via computable and indeed reasonably efficient methods. The processes described here can be implemented using technologies typically used in compilers, such as object attribution rules, they terminate for all possible inputs, and do so in polynomial time. We believe this approach will scale up to large applications, and are currently working out the details to support a tight analysis of the efficiency of the process.

This paper has explored static analysis of meta-programs to check syntactic correctness and exception closure of the generated code. Another kind of static analysis in this family, type checking of meta-programs to ensure the type correctness of the generated code, is considered by another paper in this proceedings [3].

6. Conclusions

We believe that formal models of program generation templates can support a variety of quality improvement processes that can help achieve cost-effective software reliability. This paper has presented a simple example of such a formal model and two such quality improvement processes, certification of syntactic correctness and freedom from unexpected exceptions for all programs that can be generated from a given program generation pattern. We expect the greatest advantages of this approach to be realized when it is applied to realize flexible and reliable systems in a product line approach. This approach should be augmented with systematic methods for domain analysis that culminates in the development of a domain-specific library of solutions embodied in a domain-specific software architecture that is populated with components produced by model-based software generators. When the technology matures, it should become possible for problem domain experts to specify their problem instances in terms of familiar problem domain models, and to have reliable software solutions to their problems automatically generated, without direct involvement of computer experts.

The economic advantage of this approach comes from the ability to automatically reap the benefits of each quality improvement for all past and future instantiations of the template (if past applications are regenerated). We believe that it will be profitable to explore methods for lifting many known program analysis techniques from the level of individual programs to the level of program generation patterns. This should be explored for a variety of issues that range from certifying absence of references to uninitialized variables, absence of deadlock, and many others, perhaps ultimately to template-based proof of post conditions and program termination for generated programs.

To make this vision practical, many engineering issues must be addressed, including presentation issues, methods for lightweight inference [2] and support for transforming and enhancing complex sets of analysis rules. Other issues include systematic methods for dynamic analysis, testing, and debugging of program generation rules. It is not reasonable to expect progress to occur in an instantaneous quantum leap to perfection. A realistic process is a gradual one, where simple sets of program generation rules are deployed, and gradually tuned, improved, certified, and extended. A key issue is enabling rule enhancement and exception closure extension without invalidating all previous effort on analysis and certification of the previous versions.

The difference between the program generation approach proposed here and current compiler generation tools is the associated static analysis capabilities for the program generation rules. It is possible that in the future, ultra-reliable compilers will be built using techniques derived from those introduced in this paper.

REFERENCES

1. F. Bauer, H. Ehler, A. Horsch, B. Moller, H. Partsch, O. Paukner and P. Pepper, *The Munich Project CIP.* Vol. 2: The Program Transformation System CIP-S, Springer, Berlin, 1987.

2. V. Berzins, "Light Weight Inference for Automation Efficiency", Proceedings of the *1998 ARO/ONR/NSF/DARPA Monterey Workshop on Engineering Automation for Computer Based Systems,* Monterey California, 1999.

3. N. Bjorner, "Type Checking Meta Programs", Proceedings of the Workshop on *Modeling Software System Structures in a Fastly Moving Scenario,* Santa Margherita, Italy, 2000.

4. T. Reps, *Generating Language-Based Environments*, Doctoral Dissertation, August 1982.

5. D. Volpano, R. Kieburtz, "Software Templates", CS/E 85-011, Department of Computer Science and Engineering, Oregon Graduate Center, 1985.

Domain Engineering
"Upstream" from Requirements Engineering and Software Design

Dines Bjørner
Department of Computer Science & Technology
Technical University of Denmark
DK–2800 Lyngby, Denmark
E–Mail: db@it.dtu.dk

August 22, 2000

Abstract

Before software can be developed its requirements must be stated. Before requirements can be expressed the application domain must be described. In this report we outline some of the basic facets of domain engineering.

Domains seem, it is our experience, far more stable than computing requirements, and these again seem more stable then software designs. Perhaps a way in which to more rapidly develop trustworthy software from believable requirements is to secure comprehensive domain theories.

An brief example will be given on the basis of which we briefly discuss, in this report, the notions of: domain intrinsics, domain support technologies, domain management & organisation, domain rules & regulations, domain human behaviour, etc. We show elsewhere how to "derive" requirements from domain descriptions: domain requirements: by domain projection, instantiation, extension and initialisation; interface requirements: multi–media, dialogue, etc.; and machine requirements: performance, dependability (reliability, availability, accessability, safety, etc.).

1 Background

The workshop for which this modest contribution is drawn up takes it departure point in the US President's Information Technology Advisory Committee (PITAC) 1998 interim report: *Need for software outstrip development resources. Desparately needed software is not being developed. Software must be made far more usable, reliable and powerful. Current development, test and maintenance processes must change. Scientifically sound software development approaches are required: Enabling meaningful and practical testing for consistency of specificsations and implementations.* The current document, based on [1, 2, 3, 4, 5, 6, 7, 8, 9] immodestly suggest that a stronger emphasis need be put, in future, on domain engineering as one means of reaching the PITAC goals.

2 Example: Resource Management

2.1 Synopsis and Narrative

The scope is that of resources and their management. The span is that of strategic, tactical and operations management and of actual operations. Strategic resource management is about acquiring ("expansion, upgrading") or disposing ("down-sizing, divestiture") of resources: Converting one form of resource to another. Tactical resource management is about allocating resources spatially and scheduling them for general, temporal availability. Operations resource management is about allocating resources to tasks and scheduling them for special, time interval deployment. These three kinds of resource management reflect rather different perspectives: Strategic resource management is the prerogative and responsability of executive management. Tactical resource management is the prerogative and responsability of line ("middle level") management. Operations resource management is the prerogative and responsability of operations (ie. "ground level") management.

2.2 Formalisation: Resources and their Handling

2.2.1 Formalisation: Resources

```
type R, Rn, L, T, E, A
    RS = R-set
    SR = T $\underset{m}{\rightarrow}$ RS,                  SRS = SR-infset
    TR = (T$\times$T) $\underset{m}{\rightarrow}$ R $\underset{m}{\rightarrow}$ L,   TRS = TR-set
    OR = (T$\times$T) $\underset{m}{\rightarrow}$ R $\underset{m}{\rightarrow}$ A

    A = (Rn $\underset{m}{\rightarrow}$ R-set) $\xrightarrow{\sim}$ (Rn $\underset{m}{\rightarrow}$ R-set)

value
    srm: RS $\rightarrow$ E$\times$E $\xrightarrow{\sim}$ E $\times$ (SRS $\times$ SR)
    trm: SR $\rightarrow$ E$\times$E $\xrightarrow{\sim}$ E $\times$ (TRS $\times$ TR)
    orm: TR $\rightarrow$ E$\times$E $\xrightarrow{\sim}$ E $\times$ OR

    ope: OR $\rightarrow$ TR $\rightarrow$ SR $\rightarrow$ (E$\times$E$\times$E$\times$E) $\rightarrow$ E $\times$ RS

    p: E $\rightarrow$ Bool
```

srm, trm and orm are the strategic, tactical and operations management functions. ope is the actual operations function. p is a predicate which determines whether the enterprise can continue to operate (eith its state and in its environment, e, or not.

2.2.2 Resource Formalisation — Annotation

R, L, T, E and A stand for resources, spatial locations, times, the enterprise (with its estimates, service and/or production plans, orders on hand, etc.), respectively tasks (actions). SR, TR and OR stand for strategic, tactical and operational resource views, respectively. srm, trm and orm stand for strategic, tactical, respectively operations resource management. To keep our model "small", we have had to resort to a "trick": Putting all the facts knowable and needed in order for management to function adequately into E ! E, besides the enterprise itself, also models its environment: That part of the world which affects the enterprise.

There are, accordingly, the following management functions: *Strategic resource management*, srm(rs)(e,e'''')=(e',(srs,sr)), proceed on the basis of the enterprise (e) and its current resources (rs), and "ideally estimates" all possible strategic resource possibilities (srs), and selects one, desirable (sr). The "estimation" is heuristic. Too little is normally known to compute sr algorithmically. We refer to [5] for details. *Tactical resource management*, trm(sr)(e,e'''')=(e'',(trs,tr)), proceed on the basis of the enterprise (e) and one chosen strategic resource view (sr) and "ideally calculates" all possible tactical resource possibilities (trs), and selects one, desirable (tr). As for strategic resource management, we refer to [5] for details. *Operations resource management*, orm(tr)(e,e'''')=(e''',or), proceed on the basis of the enterprise (e) and one chosen tactical resource view (tr) and effectively decides on one operations resource view (or). We refer to [5] for details. *Actual enterprise operation*, ope, enables, but does not guarantee, some "common" view of the enterprise: ope depends on the views of the enterprise its state and environment, as "passed down" by management; and ope applies, according to presecriptions kept in the enterprise state, actions, a, to named (rn:Rn) sets of resources.

2.2.3 Formalisation: Resource Handling

The above account is, obviously, rather "idealised". But, hopefully, indicative of what is going on. To give a further abstraction of the "life cycle" of the enterprise we "idealise" it as now shown:

```
value
   enterprise: RS $\overset{\sim}{\to}$ E $\overset{\sim}{\to}$ Unit
   enterprise(rs)(e) ≡
      if p(e) then
         let ($e'$,(srs,sr)) = srm(rs)(e,$e''''$),
               ($e''$,(trs,tr)) = trm(sr)(e,$e''''$),
               ($e'''$,or) = orm(tr)(e,$e''''$),
               ($e''''$,rs') = ope(or)(tr)(sr)(e,$e'$, $e''$, $e'''$) in
         let $e'''''$:E • p'(, $e''''$, $e'''''$) in
         enterprise(rs')($e'''''$) end end
      else stop end
```

The enterprise re-invocation argument, rs$'$, a result of operations, is intended to reflect the use of strategially, tatctically and operationally acquired, spatially and task allocated and scheduled resources, including partial consumption, "wear & tear", loss, replacements, etc.

An imperative version of enterprise could be:

```
value
   enterprise: E → RS → Unit
   enterprise(e)(rs) ≡
      variable ve:E := e;
      while p(ve) do
         let ($e'$,(srs,sr)) = srm(rs)(ve,$e''''$),
        $(e''$,(trs,tr)) = trm(sr)ve,$(e'''')$,
        $(e'''$,or) = orm(tr)(ve,$e'''')$,
        $(e''''$,rs$')$ = ope(or)(tr)(sr)(ve,$e', e'', e''')$ **in**
    **let** $e'''''$:E • p$'(e'''', e''''')$ **in**
    ve := $e'''''$ **end end end**
```

ope: OR → TR → SR → (E×E×E×E) → E × RS

Only the program flow of control recursion has been eliminated. The **let** e''''':E • p$'(e'''', e''''')$ **in** ... shall model a changing environment.

2.2.4 Resource Handling — Annotation

There are two forms of recursion at play here: The simple tail-recursive, next step, day-to--day recursion, and the recursive "build-up" of the enterprise state e''''. The latter is the interesting one. To solve it, by iteration towards some acceptable, not necessarily minimal fixpoint, the three levels of management and the "floor" operations change that state and "pass it around, up-&-down" the management "hierarchy". The operate function "unifies" the views that different management levels have of the enterprise, and influences their decision making. Dependence on E also models potential interaction betwen enterprise management and, conceivably, all other stake-holders. We remind the reader that we are "only" modelling the domain — with all its imperfections !

3 Discussion

The model just presented is, obviously, sketchy. But we believe it portrays important facets of domain modelling.

We are modelling a domain with all its imperfections: We are not specifying anything algorithmically; all functions are rather loosely deined, in fact only their signature is given. This means that we model well-managed as well as badly, sloppily, disastrously managed enterprises. We can, of course, define a great number of predicates on the enterprise state and its environment (e:E), and we can partially characterise intrinsics — facts that must always be true of en enterprise, no matter how well or badly it is managed. But if we "programme--specified" the enterprise then we would not be modelling the domain of enterprises, but a specifically "business process engineered" enterprise. And we would be into requirements engineering — we claim. So let us take now, a closer view of the kind of things we can indeed model in the domain !

4 Stake–holder Perspectives

There are several kind of domain stake-holders: Enterprise stake–holders: (i) owners, (ii) management: (a) executive, (b) line, and (c) "floor", managers, (iii) workers, (iv) families of the above. Non–enterprise stake–holders: (v) clients (customers), (vi) competitors, resource providers: (a) IT resource providers, (b) non–IT/non–finance resource providers, and (c) financial service providers, (vii) regulatory agencies, (viii) politicians, and (ix) the "public-at–large". For each stake-holder there usually is a distinct domain perspective: A partial specification. The example shown earlier illustrated, at some level of abstraction, the interaction between, hence the perspectives of, enterprise managers and workers, and, at a higher level of abstraction, interaction with the environment, incl. all other stake-holders.

5 Domain Facets

We shall sketch the following facets:

Domain intrinsics: That which is common to all facets.

Domain support technologies: That in terms of which most other facets (intrinsics, management, organisation, and rules & regulations) are implemented.

Domain management and organisation: That which constrains communication between enterprise stake-holders.

Domain rules & regulations: That which guides the work of enterprise stake-holders as well as their interaction and the interaction with non-enterprise stake-holders.

Domain human behaviour: The way in which domain stake-holders despatch their actions and interactions wrt. enterprise: dutifully, forgetfully, sloppily, yes even criminally.

We shall briefly characterise each of these facets.

5.1 Intrinsics

The intrinsics of a rail switch is that it can take on a number of states. A simple switch $({}^{c_|}Y_{c}^{c_/})$ has three connectors: $\{c, c_|, c_/\}$. c is the connector of the common rail from which one can either "go straight" $c_|$, or "fork" $c_/$.

$$\begin{aligned}\omega \; : \; \{ \; & \{\},\\ & \{(c,c_|)\},\{(c,c_|),(c_|,c)\},\{(c_|,c)\},\\ & \{(c,c_/)\},\{(c,c_/),(c_/,c)\},\{(c_/,c)\},\\ & \{(c,c_/),(c_|,c)\},\{(c,c_/),(c_/,c),(c_|,c)\},\{(c_/,c),(c_|,c)\} \quad \}\end{aligned}$$

Nothing is said about how a state is determined: Who sets and resets it, whether determined solely by the physical position of the switch gear, or also by visible signals up or down the rail away from the switch.

The intrinsics of a domain is a partial specification:

```
type
  Γ_i, Σ_i, Syntax, VAL_i
value
  I: Syntax → Γ_i ⥲ Σ_i ⥲ Σ_i
  V: Syntax → Γ_i ⥲ Σ_i ⥲ VAL
  E: Syntax → Γ_i ⥲ Σ_i ⥲ Σ_i × VAL_i
  D: Syntax → Γ_i ⥲ Σ_i ⥲ Γ_i × Σ_i
  C: Syntax → Γ_i ⥲ Σ_i ⥲ Γ_i × Σ_i × VAL_i
```

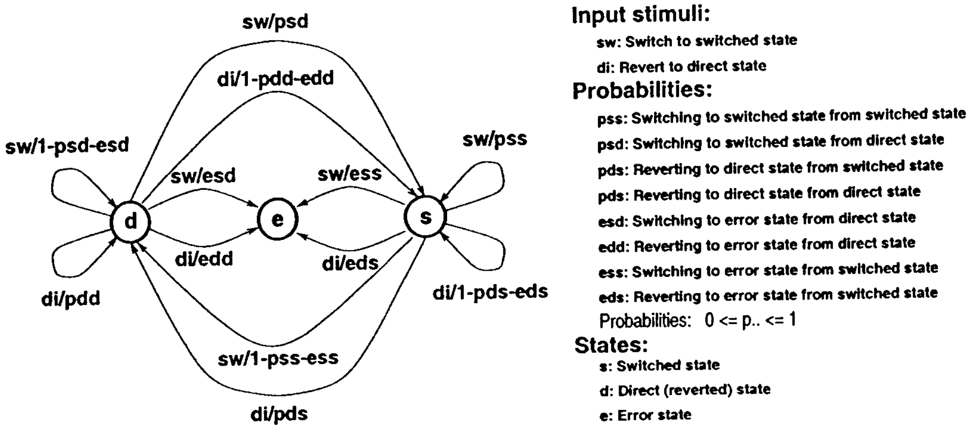
Intrinsics descriptions emphasise looseness and non-determinism.

5.2 Support Technologies

An example of different technology *stimuli:* A railway switch, "in ye olde days" of the "childhood" of railways, was *manually "thrown";* later it could be mechanically controlled from a distance by *wires and momentum "aplification"*; Again later it could be electro-mechanically controlled from a further distance by *electric signals that then activated mechanical controls*; and today switches are usually *controlled in groups that are electronically interlocked.*

An aspect of supporting technology includes the recording of state-behaviour in response to external stimuli. Figure 1 indicates a way of formalising this aspect of a supporting technology.

Figure 1: State Switching


sw/psd
di/1-pdd-edd
sw/1-psd-esd
sw/pss
sw/esd
sw/ess
d
e
s
di/edd
di/eds
di/pdd
di/1-pds-eds
sw/1-pss-ess
di/pds
Input stimuli:
sw: Switch to switched state
di: Revert to direct state
Probabilities:
pss: Switching to switched state from switched state
psd: Switching to switched state from direct state
pds: Reverting to direct state from switched state
pds: Reverting to direct state from direct state
esd: Switching to error state from direct state
edd: Reverting to error state from direct state
ess: Switching to error state from switched state
eds: Reverting to error state from switched state
Probabilities: 0 <= p.. <= 1
States:
s: Switched state
d: Direct (reverted) state
e: Error state

Support technologies "implement" contexts and states: $\gamma_i : \Gamma_i, \sigma_i : \Sigma_i$ in terms of "actual" contexts and states: $\gamma_a : \Gamma_a, \sigma_a : \Sigma_a$

```
type
  Syntax,
  Γ_i, Σ_i,  VAL_i,
  Γ_a, Σ_a,  VAL_a,
  ST = Γ_i × Σ_i ⥲ Γ_a × Σ_a
value
  sts:ST-set
  I: Syntax → Γ_a ⥲ Σ ⥲ Σ_a
  V: Syntax → Γ_a ⥲ Σ ⥲ VAL_a
  E: Syntax → Γ_a ⥲ Σ ⥲ Σ_a × VAL_a
  D: Syntax → Γ_a ⥲ Σ ⥲ Γ_a × Σ_a
  C: Syntax → Γ_a ⥲ Σ ⥲ Γ_a × Σ_a × VAL_a
```

Support technology is not a refinement. Support technology typically introduces considerations of technology accuracy, failure, etc. Axioms, not shown, characterise members of the set of support technologies sts.

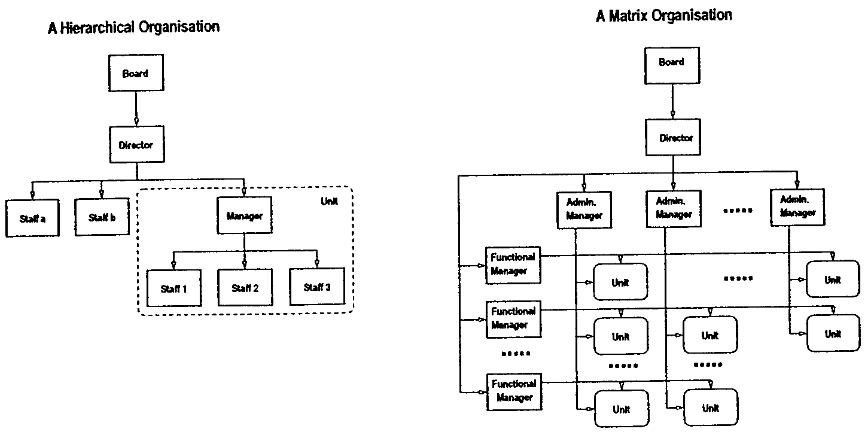
5.3 Management and Organisation

People staff the enterprises, the components of infrastructures with which we are concerned: For which we develop software. The larger these enterprises, these infrastructure components, are, the more need there is for management and organisation. The rôle of management is roughly, for our purposes, twofold: To make strategic, tactical and operational policies and see to it that they are followed, and to react to adverse conditions: Unforeseen situations, and decide upon their handling.

Policy setting should help non-management staff operate normal situations — for which no management interference is thus needed, and management "back-stops" problems: Takes these problems off the shoulders of non-management staff. To help management and staff know who's in charge wrt. policy setting and problem handling, a clear conception of the overall organisation is needed: Organisation defines lines of communication within management and staff and between these. Whenever management and staff has to turn to others for assistance they follow the command line: The paths of organigrams — the usually hierarchical box and arrow/line diagrams.

Management is a set of predicates, observer and generator functions which parameterise other, the operations functions, that is: determine their behaviour. Organisation is a set of constraints on communication behaviours.

Figure 2: Organisational Structures



```
type
   Msg, Ψ, Σ
channel
   { ms[i]:Msg | i:Sx }
value
   sys: Unit → Unit
   mgr: Ψ → in,out  { ms[i] | i:Sx } Unit
   stf: i:Sx → Σ → in,out ms[i] Unit

   sys() ≡ || { stf(i)(iσ) | i:Sx } || mgr(ψ)

   mgr(ψ) ≡
```

```
    let ψ′ = ...
        (|| { ms[i]!msg | i:Sx } ...)
        []
        ([] { let msg′ = ms[i]? in ... end | i:Sx })
          ... in mgr(ψ′) end

  stf(i)(σ) ≡
    let σ′ = ...
        (let msg = ms[i]? in f(msg)(σ) end)
        []
        (... ms[i]!msg′ ... ) ... in stf(σ′) end
```

5.4 Rules & Regulations

In China arrival and departure of trains at, respectively from railway stations are subject to the following regulation: In any three minute interval at most one train may either arrive or depart.

In many countries railway lines (between stations) are segmented into blocks or sectors. The purpose is to stipulate that if two or more trains are moving — obviously in the same direction — along the line, then there must be at least one free sector (ie. without a train) between any two such trains.

In the United State of America personal checks issued in any one state of the union must be cleared by the sending and receiving banks, if within the same state, then within 24 hours, and else within 48 or 72 hours, depending on certain further stipulated relations between the states.

```
type
  RR = Syntax → Γ → Σ → Bool
  RRS = RR-set
value
  valid: RRS → Γ → Σ → Bool
  valid(rrs)(γ)(σ) ≡
     ∀ rr:RR • rr ∈ rrs ⇒ rr(γ)(σ)
```

5.5 Human Behaviour

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

We model the "arbitrariness", the impredictability, of human behaviour by internal nondeterminism:

... b_p ⊓ b_s ⊓ b_d ⊓ b_c ...

The above shows just a fragment of a formal description of that part which reflects human behaviour. The exact, possibly deterministic, meaning of each of the b's can be separately described.

5.6 Discussion

6 Conclusion

6.1 Acknowledgements

I am grateful for the inspiration drawn from the work of Michael Jackson on requirements and software specification. I am grateful to my colleagues in IFIP WG 2.2 for like inspiration. And I am grateful to my students: Kristian Asger Eir, Ths. Hede Nielsen, M. Kalsing, et al.

6.2 Thanks

Thanks are due to US/ARO, US/NSF, US/ARO–Europe, and INDAM (GNIM) for their co-sponsorskip of this workshop. Thanks are due to Manfred Broy, LuQi, Carlo Ghezzi and Zohar Manna for co-chairing the PC. But very special and deeply warm and appreciative thanks are due to: Gianna Reggio and Egidio Astesiano of, and the University of Genoa for their indefatigable work and support in bringing us all together.

7 Bibliographical Notes

This document being a very preliminary draft lacks proper citations.

I do, however, strongly confess my delight in having studied Jackson's [10, 11, 12, 13, 14].

To support the implied claims made in the present document we refer to the following own reports and publications: [1, 2, 3, 4, 5, 6, 7, 8, 9].

References

[1] Dines Bjørner. Domain Engineering, Elements of a Software Engineering Methodology — Towards Principles, Techniques and Tools — A Study in Methodology. Research report, Dept. of Computer Science & Technology, Technical University of Denmark, Bldg. 343, DK–2800 Lyngby, Denmark, 2000. One in a series of summarising research reports [2, 3].

[2] Dines Bjørner. Requirements Engineering, Elements of a Software Engineering Methodology — Towards Principles, Techniques and Tools — A Study in Methodology. Research report, Dept. of Computer Science & Technology, Technical University of Denmark, Bldg. 343, DK–2800 Lyngby, Denmark, 2000. One in a series of summarising research reports [1, 3].

[3] Dines Bjørner. Software Design: Architectures and Program Organisation, Elements of a Software Engineering Methodology — Towards Principles, Techniques and Tools — A Study in Methodology. Research report, Dept. of Computer Science & Technology, Technical University of Denmark, Bldg. 343, DK–2800 Lyngby, Denmark, 2000. One in a series of summarising research reports [1, 2].

[4] Dines Bjørner, Souleymane Koussoube, Roger Noussi, and Gueorgui Satchok. Jackson's Problem Frames: Domain, Requirements and Design. In Shaoying Liu, editor, *International Conference on Formal Engineering Methods: ICFEM'97*, Washington D.C., USA, 12–14 November 1997. IEEE Computer Science Press; IEEE sponsored conference, Hiroshima, Japan.

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

[6] Dines Bjørner. Pinnacles of Software Engineering: 25 Years of Formal Methods. *Annals of Software Engineering*, 2000. Eds. Dilip Patel and Wang Yi.

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

[8] Dines Bjørner. Where do Software Architectures come from ? Systematic Development from Domains and Requirements. A Re–assessment of Software Engneering ? *South African Journal of Computer Science*, 1999. Editor: Chris Brink.

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

[10] Michael A. Jackson. Problems, methods and specialisation. *Software Engineering Journal*, pages 249–255, November 1994.

[11] Michael A. 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.

[12] Michael A. 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. .

[13] Michael A. Jackson. The meaning of requirements. *Annals of Software Engineering*, 3:5–21, 1997.

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

The Partial Spechilada

Nikolaj S. Bjørner
bjorner@kestrel.edu
Kestrel Institute

1 Introduction

This paper examines different ways to represent and validate core program transformations. As the leading example, we take finite differencing, and subject it to alternative encodings. Our first encoding is to represent finite differencing equationally, and appeal to rewrite based simplification to apply it. The second encoding is to represent the transformation as a library refinement and use *diagram pushouts* to apply it. Finally we consider an encoding as a meta-program that works on the abstract syntax tree of Specware terms. While meta-programming allows finegrained control over the transformations it may be less declarative than object level axiomatization, and often less transparent. The assurance problem for meta-programs becomes an issue when having to rely on a *dynamically growing set of transformations, in a fastly moving scenario.* While this can be recast as a compiler correctness problem, we shall discuss type based partial correctness criteria, an even more interestingly, frameworks for automatic inference of meta-typeability: the meta programs transform well typed programs into well typed programs.

The exposition is technical, although we only present elementary concepts and examples to ease the readability. The high points of this paper can be summarized as equation (4) and Figure 9.

2 Specware and Designware

Kestrel has for a number of years been developing and researching tools for program specification and synthesis. The Kestrel Institute Development System, KIDS, is a stable program transformation system. While highly effective at its design goals, it does not support data type refinement. The more recent tool Specware is based on concepts from category theory and addresses this point, while in itself does not deal with program transformation. On top of Specware we are developing program transformation capabilities. This extension is called Designware.

3 Transformations as Equations

Finite differencing, or strength reduction, is a program transformation that incrementally computes intermediate values in loops. A simple example is the function

$$\mathrm{sosq}(n : \mathsf{Nat}, m : \mathsf{Nat}) \;=\; \textbf{if } \; n \geq m \textbf{ then } \; 0 \textbf{ else } \; n^2 + \mathrm{sosq}(n+1, m) \quad (1)$$

that when supplied with arguments n and m computes the sum of squares from n to $m-1$. Since

$$(n+1)^2 = n^2 + 2 \cdot n + 1$$

it is possible to avoid computing $(n+1)^2$ by remembering the value of n^2 and then adding $2 \cdot n + 1$, which itself can be computed using a left shift and a bitwise or with 1. Thus, the program

$$\begin{aligned} \mathrm{sosq}(n : \mathsf{Nat}, m : \mathsf{Nat}) &= \mathrm{sosq}'(n^2, n, m) && (2) \\ \mathrm{sosq}'(n_{sq}, n, m) &= \textbf{if } \; n \geq m \textbf{ then } \; 0 \textbf{ else} \\ & \qquad n_{sq} + \mathrm{sosq}'(n_{sq} + \mathit{lshift}(n) + 1, n+1, m) && (3) \end{aligned}$$

avoids multiplication entirely in the main loop. We can also make sosq' tail-recursive in a further optimization based on the fact that + is commutative. Of course, this example is inherently artificial since sosq is computable by a closed term involving cubes. Nevertheless, not every loop is reducible in this way, and finite differencing has a distinguished place as a useful program transformation.

3.1 An equational characterization of finite differencing

The use of finite differencing is pervasive in program transformation systems such as KIDS, Hylo, RAPS, and Designware, among others.

In essence, finite differencing adds extra arguments to a recursive function that pre-calculate selected subterms. Finite differencing improves the program if the new arguments are less expensive to update in recursive calls than the old ones. We can apply finite differencing in three steps:

1. Select a subterm to be replaced by an induction variable.
2. Generate the finitely differenced version of the recursive function.
3. Simplify the resulting function using constraints on the induction variable.

Using higher-order functions, we can characterize steps 1 and 2 using a single equation:

$$\begin{array}{l} \forall X, Y \;.\; \mathbf{fix}(\lambda f\ a.\ X\ (Y\ a)\ f\ a) = \\ \qquad \lambda a.\mathbf{fix}(\lambda g\ (b,c).X\ c\ (\lambda\ d.\ g\ (d,\ Y\ d))\ b)\ (a,\ Y\ a) \end{array} \;. \tag{4}$$

With type annotations, this equation reads:

$$\begin{array}{l} \forall X : \gamma \to (\alpha \to \beta) \to \alpha \to \beta, \\ \forall Y : \alpha \to \gamma \\ \quad \mathbf{fix}\left(\begin{array}{l} \lambda f : \alpha \to \beta,\ a : \alpha. \\ \quad X\ (Y\ a)\ f\ a \end{array} \right) \quad = \\ \quad \lambda a : \alpha.\mathbf{fix}\left(\begin{array}{l} \lambda g : \alpha \times \gamma \to \beta,\ (b,c) : \alpha \times \gamma. \\ \qquad X\ c\ (\lambda\ d : \alpha.\ g\ (d,\ Y\ d))\ b \end{array} \right)\ (a,\ Y\ a) \end{array} \tag{5}$$

where **fix** is the fixpoint operator of type $\forall \alpha \;.\; (\alpha \to \alpha) \to \alpha$ and satisfies $\mathbf{fix}\ f = f(\mathbf{fix}\ f)$.

In fact, we will use subtypes to strengthen the right hand side with the additional information that the λ-bound c satisfies the invariant $c = (Y\ b)$. Thus, the right hand side takes the form

$$\begin{array}{l} \lambda a : \alpha.\mathbf{fix}\left(\begin{array}{l} \lambda g : \delta \to \beta,\ (b,c) : \delta. \\ \qquad X\ c\ (\lambda\ d : \alpha.\ g\ (d,\ Y\ d))\ b \end{array} \right)\ (a,\ Y\ a) \\ \qquad\qquad\qquad\qquad \text{where} \quad \delta = \{(b,c) \mid Y\ b = c\} \end{array}$$

Notice, however, that we have left the subtype coercions implicit, as these can be inferred and checked automatically.

3.2 Calculating sosq'

Returning to the sum of squares example, we present the original program in a form suitable for applying equation (5):

$$\mathrm{sosq} \;=\; \mathbf{fix}\left(\begin{array}{l} \lambda \mathrm{sosq} : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat},\ (n,m) : \mathsf{Nat} \times \mathsf{Nat}. \\ \quad X\ (Y\ a)\ f\ a \end{array} \right) \tag{6}$$

$$X \;=\; \lambda n_{sq} \;.\; \lambda \mathrm{sosq} \;.\; \lambda(n,m) \;.\; \left(\begin{array}{l} \mathbf{if}\ \ n \geq m\ \mathbf{then}\ \ 0\ \mathbf{else} \\ \qquad n_{sq} + \mathrm{sosq}(n+1, m) \end{array} \right) \tag{7}$$

$$Y \;=\; \lambda(n,m) \;.\; n^2 \tag{8}$$

After rewriting using equation (5), we have

$$\lambda(n,m) : \mathsf{Nat} \times \mathsf{Nat} \;.\; \mathrm{sosq}'\ ((n,m),\ n^2)$$

where

$$\mathrm{sosq}' \;=\; \mathbf{fix}\left(\begin{array}{l}\lambda \mathrm{sosq}', ((n,m),\ n_{sq}).\\ \qquad\qquad X\ n_{sq}\ (\lambda d\ .\ \mathrm{sosq}'\ (d,\ Y\ d))\ (n,m)\end{array}\right) \tag{9}$$

Simplifying the inner expression, we obtain precisely the version of sosq' from equation (3):

$$\begin{array}{lll}
 & X\ n_{sq}\ (\lambda d\ .\ \mathrm{sosq}'\ (d,\ Y\ d))\ (n,m) & (10)\\
= & \quad \{\text{by } \beta \text{ expansion}\} & \\
 & \mathbf{if}\ \ n \geq m\ \mathbf{then}\ \ 0\ \mathbf{else}\ n_{sq} + \mathrm{sosq}'\ ((n+1,m),\ (n+1)^2) & (11)\\
= & \quad \{\text{by the rewrites } (x+1)y = xy+y, x(y+1) = xy+x\} & \\
 & \mathbf{if}\ \ n \geq m\ \mathbf{then}\ \ 0\ \mathbf{else}\ n_{sq} + \mathrm{sosq}'\ ((n+1,m),\ (n^2+2\cdot n+1)) & (12)\\
= & \quad \{\text{by applying the subsort property on } n_{sq}\} & \\
 & \mathbf{if}\ \ n \geq m\ \mathbf{then}\ \ 0\ \mathbf{else}\ n_{sq} + \mathrm{sosq}'\ ((n+1,m),\ (n_{sq}+2\cdot n+1)) & (13)
\end{array}$$

Specware has a rewrite-based simplification engine that uses higher-order matching. When simplifying terms in context, subtype information, such as $n_{sq} : \{x : \mathsf{Nat} \mid n^2 = x\}$, is used to supply the rewrite engine with auxiliary rewrites (in our example $n^2 = n_{sq}$). Thus, when given the equalities (7) and (8), and the hint that $(n+1)^2 = n^2+2n+1$, the rewrite engine rewrites (10) into (13). This is illustrated later in Figure 7.

3.3 Applying finite differencing

Similarly, the higher-order rewrite engine can perform finite differencing by matching a term against the left hand side of equality (5) and replacing it with the right hand side of (5). In this connection, X and Y act as meta variables that can be bound to arbitrary closed subterms, as in equations (7) and (8).

Unfortunately, a single application of higher-order matching may return several matching substitutions. Thus, we need to consider how to choose which substitution to use. Of course, in our example, there is only one other substitution, and it is not very interesting: $Y' = \lambda(n,m)\ .\ n$ and X' is X except that we replace n_{sq} by n_{sq}^2.

To show the correctness of the equational presentation of finite differencing, we simply need to verify equation (5). The proof uses fixpoint induction and the monotonicity properties of $\sqsubseteq$. In this context, fixpoint induction is the rule:

$$\boxed{\begin{array}{l}
M\ \bot \sqsubseteq N\\
\underline{\forall Z\ .\ M\ Z \sqsubseteq N \ \ \Rightarrow\ \ M\ (M\ Z) \sqsubseteq N}\\
\mathbf{fix}\ M \sqsubseteq N
\end{array}}$$

For example:

$$\begin{array}{ll}
& (\lambda f\ a.\ X\ (Y\ a)\ f\ a)\ \bot \\
= & \qquad \{\text{by } \beta \text{ reduction}\} \\
& \lambda a.\ X\ (Y\ a)\ \bot\ a \\
\sqsubseteq & \qquad \{\text{since } \bot \sqsubseteq X \text{ for every } X\} \\
& \lambda a.\ X\ (Y\ a)\ (\lambda\ d.\ \mathsf{G}\ (d,\ Y\ d))\ a \\
= & \qquad \{\text{by folding with the definition of } \mathbf{fix}\} \\
& \lambda a.\ \underbrace{\mathbf{fix}(\lambda g\ (b,\ c).X\ c\ (\lambda\ d.\ g\ (d,\ Y\ d))\ b)}_{\mathsf{G}}\ (a,\ Y\ a)
\end{array}$$

$$\begin{array}{ll}
& (\lambda f\ a.\ X\ (Y\ a)\ f\ a)\ ((\lambda f\ a.\ X\ (Y\ a)\ f\ a)\ Z) \\
= & \qquad \{\text{by } \beta \text{ contraction}\} \\
& \lambda a.X\ (Y\ a)\ ((\lambda f\ a.\ X\ (Y\ a)\ f\ a)\ Z)a \\
\sqsubseteq & \qquad \{\text{by the induction hypothesis}\} \\
& \lambda a.X\ (Y\ a)\ (\lambda a\ .\mathsf{G}(a, Y\ a))a \\
= & \qquad \{\text{by abstracting } a \text{ and } Y\ a\} \\
& \lambda a.(\lambda(b,c)\ .\ X\ c\ (\lambda a\ .\mathsf{G}(a, Y\ a))b)(a, Y\ a) \\
= & \qquad \{\text{by folding the definition of } \mathsf{G}\} \\
& \lambda a.\ \underbrace{\mathbf{fix}(\lambda g\ (b,\ c).X\ c\ (\lambda\ d.\ g\ (d,\ Y\ d))\ b)}_{\mathsf{G}}\ (a,\ Y\ a)
\end{array}$$

4 Transformations as pushouts

In this section, we describe Designware's framework for program transformation. Designware is a system developed at Kestrel that uses category theory as a tool for organizing program development. In Designware, transformations on specifications (and, in particular, programs) are realized using *pushouts.* Naturally, we use finite differencing as an example to illustrate pushouts.

4.1 Specifications

Consider the two specifications **FD-Source** and **FD-Target** below in Figure 1 and 2.

4.2 Morphisms

We can relate **FD-Source** and **FD-Target** using a *morphism.* For this discussion, we define a morphism as a mapping from sort symbols in the source to sort terms in the target and from op symbols in the source to (closed) op terms in

```
spec FD-Source =
  sort $\alpha, \beta, \gamma$
  op $X : \gamma \to (\alpha \to \beta) \to \alpha \to \beta$
  op $Y : \alpha \to \gamma$
  def $f(a) = X\ (Y\ a) f a$
end-spec
```

Figure 1: Finite differencing source

```
spec FD-Target =
  sort $\alpha, \beta, \gamma$
  op $X : \gamma \to (\alpha \to \beta) \to \alpha \to \beta$
  op $Y : \alpha \to \gamma$
  def $f'(b, c) = X\ c\ (\lambda\ d.\ f'(d, Y\ d))\ b$
  def $f(a) = f'(a, Y\ a)$
end-spec
```

Figure 2: Finite differencing target

the target. Thus, a morphism shows how to translate sorts and operations from one specification to another. Furthermore, the resulting translation is required to map axioms in the source to theorems in the target. Morphisms are also called *refinements*; a morphism from theory A to theory B shows how B is a refinement of A.

In our example, we can relate **FD-Source** to **FD-Target** using the morphism

$$\alpha \mapsto \alpha,\ \beta \mapsto \beta,\ \gamma \mapsto \gamma,\ X \mapsto X,\ Y \mapsto Y,\ f \mapsto f$$

The source specification contains one axiom, namely the equality $f(a) = X\ (Y\ a)\ f\ a$. This equality is provable using the definitions in the target, as we saw in Section 3.3.

This morphism can be stored in a library and (re)used to apply finite differencing. In our example, we begin with this specification:

```
spec Sosq =
  def $\mathrm{sosq}(n, m) =$ if $n \geq m$ then $0$ else $n^2 + \mathrm{sosq}(n+1, m)$
end-spec
```

Figure 3: Sum of squares

Then, using higher-order matching, we construct a morphism from **FD-Source** to **Sosq**. The morphism we are interested in is not surprisingly characterized by equations (7) and (8).

4.3 Pushouts

Given the morphism from **FD-Source** to **FD-Target** and the morphism from **FD-Source** to **Sosq**, we perform a *pushout* to compute $\textbf{Sosq}_{FD}$, the finitely differenced sum of squares:

$$\begin{tikzcd}
\text{Library refinement} & \textbf{FD-Source} \arrow[d] \arrow[r] & \textbf{Sosq} \arrow[d, dotted] & \text{Thread of refinement} \\
& \textbf{FD-Target} \arrow[r, dotted] & \textbf{Sosq}_{FD} \arrow[d] & \\
& & \textbf{Simplified-Sosq}_{FD} &
\end{tikzcd}$$

The resulting specification $\textbf{Sosq}_{FD}$ contains equation (3). The pushout operation computes $\textbf{Sosq}_{FD}$ and the two dashed morphisms into it automatically. In fact, $\textbf{Sosq}_{FD}$ is the least constrained specification that admits morphisms from **FD-Target** and **Sosq** and that makes the above square commute.

After applying finite differencing, we can apply additional simplification steps to construct a simplified specification $\textbf{Simplified-Sosq}_{FD}$ and a morphism from $\textbf{Sosq}_{FD}$ to it.

```
spec  Simplified-Sosq-FD =
   def sosq($n, m$) = sosq'(($n, m$), $n^2$)
   def sosq'(($n, m$), $n_{sq}$) =  if  $n \geq m$ then  0 else
                                        $n_{sq}$ + sosq'(($n + 1, m$), $n_{sq} + 2 \times n + 1$)
end-spec
```

Figure 4: Finitely differenced, simplified, Sum of squares

4.4 Summary

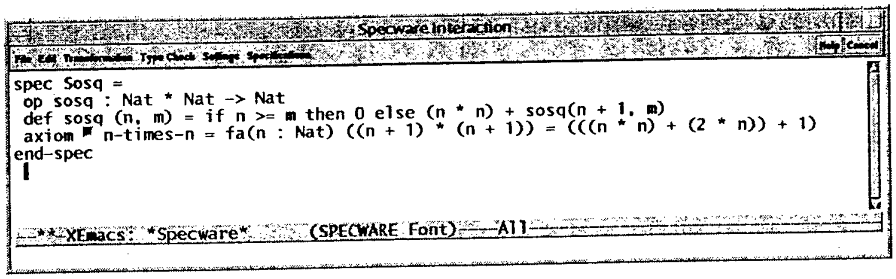
This section presented the basic concepts of the Designware framework: specifications, morphisms, and pushouts. Designware uses morphisms to represent both program transformations and data type refinements. To verify a transformation, we check that the morphism representing it indeed sends axioms to theorems. To apply a transformation, we first find a morphism from the source of the transformation morphism to the specification to be transformed using higher-order matching or witness finding. Then we compute the pushout of this morphism and the transformation morphism. The pushout operation can be implemented efficiently using the standard algorithm for union-find.

5 Transformations as Meta Programs

So far, we have examined two approaches to applying program transformations. The first approach applied transformations using a rewriter based on higher-order matching. The second approach applied transformations by pushout in the category of specifications and morphisms. Both styles required higher-order matching to determine where to apply the transformation.

Common transformations may be encoded more conveniently using meta programs, that is, as programs that manipulate other programs. For example, in the actual Designware implementation, finite differencing is performed by a specialized tactic. The user selects a subterm occurring in some definition, and the tactic creates a new definition with the subterm added as an extra argument. This process does not require any higher-order matching or pushout computation. Instead, the finite differencing transformation is hardwired as a meta-program.

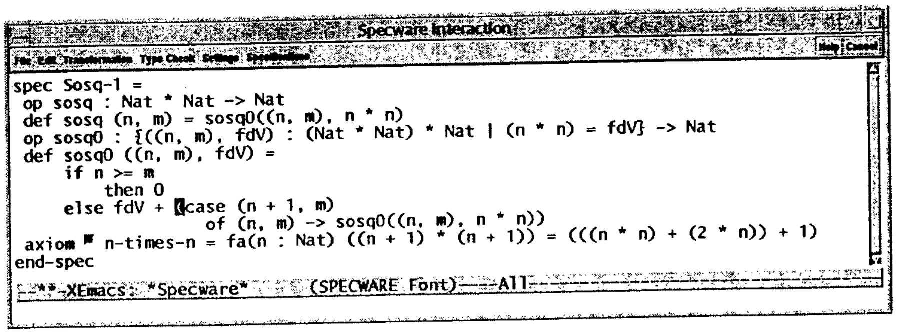
The effect of applying the finite differencing tactic, followed by simplification is illustrated in Figures 5, 6, and 7.



```
spec Sosq =
 op sosq : Nat * Nat -> Nat
 def sosq (n, m) = if n >= m then 0 else (n * n) + sosq(n + 1, m)
 axiom n-times-n = fa(n : Nat) ((n + 1) * (n + 1)) = (((n * n) + (2 * n)) + 1)
end-spec

--**-XEmacs: *Specware*      (SPECWARE Font)----All----------------
```

Figure 5: Original sum of squares specification.

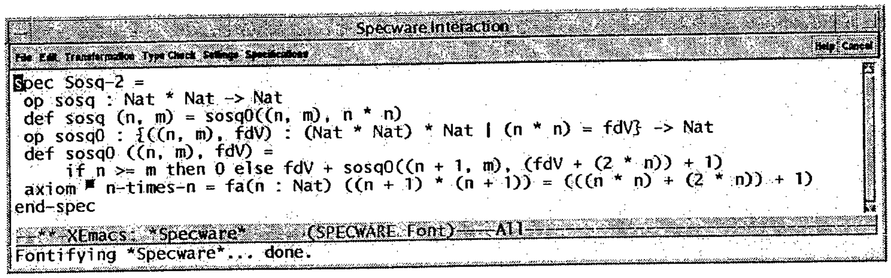


```
spec Sosq-1 =
 op sosq : Nat * Nat -> Nat
 def sosq (n, m) = sosq0((n, m), n * n)
 op sosq0 : {((n, m), fdV) : (Nat * Nat) * Nat | (n * n) = fdV} -> Nat
 def sosq0 ((n, m), fdV) =
     if n >= m
        then 0
     else fdV + (case (n + 1, m)
                   of (n, m) -> sosq0((n, m), n * n))
 axiom n-times-n = fa(n : Nat) ((n + 1) * (n + 1)) = (((n * n) + (2 * n)) + 1)
end-spec
--**-XEmacs: *Specware*      (SPECWARE Font)----All----------------
```

Figure 6: Finitely differenced sum of squares specification.

Before presenting the meta program, let us re-examine **FD-Target**:

$$\textbf{def}\ f'(b,c) = X\ c\ (\lambda\ d.\ f'(d, Y\ d))\ b$$



```
Spec Sosq-2 =
 op sosq : Nat * Nat -> Nat
 def sosq (n, m) = sosq0((n, m), n * n)
 op sosq0 : {((n, m), fdV) : (Nat * Nat) * Nat | (n * n) = fdV} -> Nat
 def sosq0 ((n, m), fdV) =
     if n >= m then 0 else fdV + sosq0((n + 1, m), (fdV + (2 * n)) + 1)
 axiom n-times-n = fa(n : Nat) ((n + 1) * (n + 1)) = (((n * n) + (2 * n)) + 1)
end-spec
--**-XEmacs: *Specware*      (SPECWARE Font)----All------------------------
Fontifying *Specware*... done.
```

Figure 7: Simplified sum of squares specification.

$$\textbf{def } f(a) = f'(a, Y\ a)$$

In this context we can rewrite the definition of f', first by α-renaming, then by folding with the definition of f.:

$$\begin{aligned} f'(a,c) &= X\ c\ (\lambda\ a.\ f'(a, Y\ a))\ a && (14)\\ &= X\ (Y\ a)(\lambda\ a.\ f'(a, Y\ a))\ a && (15)\\ &= X\ (Y\ a)\ f\ a && (16)\end{aligned}$$

Apparently we have obtained nothing, as this is almost the original definition of f, without the induction variable c. On the other hand, this equation suggests a simple way to implement finite differencing: introduce an auxiliary function f' with an argument c that satisfies $Ya = c$, make the body of f' be the body of f and let f call f' as done in **FD-Target**. In the resulting specification, replace every use of f by $\lambda a.\ f'(a, Y\ a)$ (this corresponds to unfolding f by its immediate definition). We can also simplify the body of f' using the rewrite $Y\ a = c$.

This yields Specware's actual implementation of finite differencing. Specifically, suppose that we have selected a subterm N in a definition $f(a) = M$ of a specification *spc*. We pass f, a, N, M and *spc* as arguments to the following finite differencing transformation.

```
def fd($f, a, N, M, spc$) =
    let
        $f' = \text{fresh}()$
        $b = \text{fresh}()$
        $spc = spc \dagger [f \mapsto \lceil \lambda a.f'(a, N)\rceil] \dagger [f' \mapsto \lceil \lambda(a, b).M\rceil]$
        $unfold = \lambda x$ . if $x = f$ then $\lceil \lambda a.\ f'(a, N)\rceil$ else $x$
    in
        $mapSpec\ unfold\ spc$
```

Figure 8: A meta program for finite differencing

In this tactic, we use Quine brackets to distinguish object from meta-level terms. The auxiliary function *mapSpec* maps the function *unfold* across every subterm occurring in the specification, and the map update function † overrides *spc* with a new definition.

5.1 Type correctness

Ideally, we would like to derive the above meta program for finite differencing automatically from equation 5. However, for the moment, we are willing to settle for an automatic proof that the above meta program does not introduce type errors into the programs it transforms. It turns out that we can provide this guarantee by type checking the meta program in an expressive, two-level type system.

Ordinarily, meta programs represent all object terms, independent of their object type, as values of a single meta type Term. That is, a term of object type Nat is represented as a value of meta type Term, and similarly for a term of object type String. The obvious solution is to split the single meta type Term into a family of meta types Term[s] indexed by their object level types. Thus, an object term of type Nat is represented as a value of meta type Term[Nat]. We also introduce types Var[s] and *opSorts* to represent variables and operations.

Using this type system, which can be embedded into the system LF, we can type the finite differencing meta program as shown in Figure 9.

```
def fd($f : \mathsf{Op}[\alpha \to \beta], a : \mathsf{Var}[\alpha], N : \mathsf{Term}[\gamma], M : \mathsf{Term}[\beta], spc$) =
    let
        $f' : \mathsf{Op}[\{(a,b) : \alpha \times \gamma \mid N = b\} \to \beta] = \mathrm{fresh}()$
        $b : \mathsf{Var}[\gamma] = \mathrm{fresh}()$
        $spc = spc \dagger [f \mapsto \ulcorner \lambda a.f'(a,N) \urcorner] \dagger [f' \mapsto \ulcorner \lambda (a,b).M \urcorner]$
        $unfold : \forall \alpha\ .\ \mathsf{Term}[\alpha] \to \mathsf{Term}[\alpha]$
                    $= \lambda x$ . if $x = f$ then $\ulcorner \lambda a.f'(a,N) \urcorner$ else $x$
    in
        $mapSpec\ unfold\ spc$
```

Figure 9: A type annotated meta program for finite differencing

Let's examine the typing of the *map* operator. *Map f t* applies f to each subterm of t. If object terms are formed using the constructors App, Lam, Var and Op, we can define *map* as:

$$\begin{aligned} map\ f\ (\mathsf{App}\ M\, N) &= f(\mathsf{App}\ (map\ f\ M)\,(map\ f\ N)) \\ map\ f\ (\mathsf{Lam}\ v\,.\,M) &= f(\mathsf{Lam}\ v\,.\,map\ f\ M) \\ map\ f\ (\mathsf{Var}\, v) &= f(\mathsf{Var}\, v) \\ map\ f\ (\mathsf{Op}\, g) &= f(\mathsf{Op}\, g) \end{aligned}$$

Map then has the interesting type:

$$(\forall\alpha \;.\; \mathsf{Term}[\alpha] \rightarrow \mathsf{Term}[\alpha]) \quad\rightarrow\quad (\forall\beta \;.\; \mathsf{Term}[\beta] \rightarrow \mathsf{Term}[\beta]) \;.$$

This type is not expressible in the standard Hindley-Milner polymorphic type system since the scope of the type variables α and β is not the whole term. However, this type *is* expressible in the system of rank-2 bounded polymorphism. Type inference for rank-2 bounded polymorphism is decidable *until* we allow recursive functions, such as *map*, at which point it becomes undecidable. On the other hand, this example applies rank-2 polymorphism to an interesting problem; it isn't contrived or pathological in the least. For further technical details, see [1].

In this connection, a noteworthy approach that allows more declarative style meta-programming by adding higher-order matching is described in [2].

5.2 Bound and free variables

A fundamental limitation of the above approach is that the typing $M : \mathsf{Term}[s]$ tells us nothing about the free variables of M. For example, the finite differencing transformation only makes sense if the only free variable in N is a. The above type system still allows us to type check meta-programs that expose bound variables and capture free variables.

Fortunately, this problem is solved in [3], which presents a calculus that includes automatic α-renaming of bound variables. The calculus forces α-renaming into the language implementation but in return provides a type system for checking that programs do not expose bound variables or capture free variables. Our example can with a few modifications be fed to this system and thus certified to be safe with respect to the standard variable scoping rules. The challenge remains to develop a type system that handles variable scoping without building in α-renaming.

6 Conclusion

We used the finite-differencing program to expose different ways to encode program transformation tactics. While the framework of Designware is general enough to accomodate tactics such as finite differencing as a declaratively given library refinement, we, at least in practice, are not making use of this, but instead encode it using a hardwired tactic written as a meta-program. The validation problem for meta-programs involves reasoning at two levels, which is clearly more complicated than proving theorems that only deal with one level. Luckily, unguided support for, partial correctness, such as meta-type safety, can be addressed using rank-2 polymorphism.

Acknowledgements I would like to thank the workshop organizers for giving me an opportunity to summarize thoughts otherwise not possible in my daily work. I would also like to thank my colleagues at Kestrel Institute for a joy-full environment. Without David Espinosa, we would not have had the most elegant category theory-based bootstrapped synthesis system (this is a highly competitive field). He translated this paper from Danish to English. Many thanks also to Cordell Green for his subtle humor and to Dusko Pavlovic for his equally unsubtle personality. Doug Smith's steady, consistent, and always energetic drive stands out as the most stimulating source for new, interesting and challenging, projects. John Anton is always around to give his support and do the heavy lifting. As usual, this note had not been written without a good dosis of Richard Waldinger's favorite Dark Roasted French Sumatra coffee. His 24 hour feedback service on using SNARK in connection with Specware is simply amazing.

References

[1] N. Bjørner. Type checking meta-programs. In *LFM'99: Proceedings of Workshop on Logical Frameworks and Meta-languages*, 1999.

[2] Z. Hu and M. Takeichi. Calculation carrying programs. Technical Report METR 99-07, University of Tokyo, September 1999.

[3] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction, MPC2000, Proceedings, Ponte de Lima, Portugal, July 2000*, volume ? of *Lecture Notes in Computer Science*, pages ?–? Springer-Verlag, Heidelberg, 2000.

Dynamic Distributed Systems*

Towards a Mathematical Model

Extended Abstract

Manfred Broy
Institut für Informatik, Technische Universität München
D-80290 München, Germany

Abstract. This paper aims at a mathematical foundation of flexible information processing architectures that support the mobility and dynamics of systems by a formal system model.

Index terms. Software Engineering, System Models, Formal Methods, Mobility, Dynamic Systems

1. Introduction

In this paper we work out a theoretical foundation for dynamic systems architectures. For information processing systems of today and of tomorrow *dynamics* and *mobility* are key issues. We are aiming at a formal model in this paper that allows us to give precise definitions for key notions that arise in this context the *dynamics of nets* and the *dynamics of distributed systems*. We are interested in a precise definition, description and in the mathematical foundation of these notions and, moreover, in modeling the following technical key concepts in system structures:

- a *net* models a distributed system of components interacting in parallel and connected by communication channels,
- a component is called *dynamic*, if it can change its (syntactic) interface consisting of its active input and output channels,
- a net is called *dynamic* if it changes its structure (its set of existing components and its set of channels) during its lifetime, otherwise it is called *static*,

In the following we formalize the notions introduced above. We give a mathematical model that allows us to capture the mentioned aspects.

2. Component Models: Interface Models by Streams

A (system) *component* is an active information processing unit that communicates with its environment through a set of input and output channels. This communication takes place in a (discrete) time frame.

*) Part of this work was carried out within the Forschungsverbund ForSoft, sponsored by the Bayerische Forschungsstiftung.

As a basic model for the behavior of system components we use relations on timed streams[1]. We model the time flow in systems by a sequence of time intervals. Let M be a set of elements called messages. Timed streams model communication histories for communication channels by an infinite sequence of finite sequences of messages (for a rigorous treatment see [Broy 95]). Each finite sequence represents the sequence of messages communicated within a particular time interval. On the basis of this simple model we are able to introduce a quite flexible notation that we will use throughout this paper in specifications and for our models.

By M^∞ we denote the set of infinite sequences of elements of the set M, which can be represented by functions $\mathbb{N}\backslash\{0\} \rightarrow M$, by M^* we denote the set of finite sequences of elements from M. Using this notation, by

$$(M^*)^\infty$$

we denote the set of infinite sequences of finite sequences, which can be represented by functions $\mathbb{N}\backslash\{0\} \rightarrow M^*$, also called streams of finite sequences of elements of the set M.

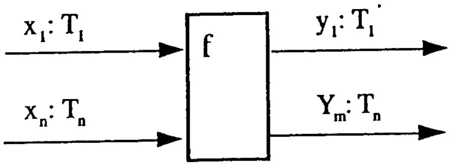


Fig. 1 Graphical representation of a component as a data flow node with input channels $x_1, \dots, x_n$ and output channels $y_1, \dots, y_m$ and their respective types

Let I be the set of input channels and O be the set of output channels. Formally a channel is nothing but an identifier. With every channel in the channel set $I \cup O$ we associate a data type indicating the type of messages sent on that channel. For simplicity, in the following we use uniformly only one set of messages denoted by M representing the data types for the messages on the channels to keep the mathematics more readable. Our approach, however generalizes to individually typed channels in a straightforward way.

By the pair (I, O) the *syntactic interface* of a system component is represented. A graphical representation of a component with its syntactic interface and individual channel types is shown in Fig. 1.

We describe the *black box behavior* of a component by an *I/O-function*. It represents a relation between the input streams and the output streams of a component that fulfills certain conditions with respect to their timing. An I/O-function is a set-valued function on valuations of the input channels by timed streams. The function yields a set of histories for the output channels for every input history. This way, an I/O-function is a function

$$F: (I \rightarrow (M^*)^\infty) \rightarrow \wp(O \rightarrow (M^*)^\infty)$$

which fulfills the following *timing property*. The timing property axiomatises the time flow and reads as follows:

[1] Another option is to use nontimed streams, especially, when dealing with systems where time is not an issue. However, even for these systems it is often convenient to be able to talk about time, especially, when combining such systems with time dependent components.

$$x{\downarrow}t = z{\downarrow}t \Rightarrow \{y{\downarrow}t{+}1: y \in F(x)\} = \{y{\downarrow}t{+}1: y \in F(z)\}$$

For a stream s, s↓t denotes the sequence that is the prefix of the stream s and contains t finite sequences. In other words, s↓t denotes the communication history of s until time t. This operation is extended to histories in $C \to (M^*)^\infty$, where C is a set of channels, pointwise. The timing property expresses that the set of possible output histories for the first t+1 time intervals only depends on the input histories for the first t time histories. In other words, the processing of messages in a component takes at least one tick of time. This way causality between input and output is guaranteed. We call functions with this property *time-guarded* or *strongly causal.*

By COM[I, O] we denote the set of all strongly causal I/O-functions with the syntactic interface (I, O), that is, with the set of input channels I and the set of output channels O. For each F ∈ COM[I, O], In(F) = I denotes its set of input channels and Out(F) = O denotes its set of output channels. By COM we denote the set of all strongly causal I/O-functions, also called data flow behaviors.

3. A Mathematical Model of Data Flow Nets

We model distributed systems by data flow nets. Let K be a set of identifiers for components[1] (represented by data flow nodes) and O be a set of output channels. A distributed system (ν, O) in the form of a data flow net with the syntactic interface (I, O) is represented by the mapping

$$\nu: K \to \mathrm{COM}$$

that associates with every node labeled by the identifier k ∈ K a component behavior (an interface behavior given by an I/O-function)[2]. In principle, we can think about the component associated with an identifier as its type (or its class). This way we are very close to object orientation.

The set

$$I = (O \cup \{c \in In(\nu(k)): k \in K\}) \backslash \{c \in Out(\nu(k)): k \in K\}$$

denotes the set of input channels of the net. As a well-formedness condition we require that for all component identifiers k, j ∈ K (with $k \neq j$) the sets of output channels of the components ν(k) and ν(j) are disjoint. This is formally expressed by the equation $Out(\nu(k)) \cap Out(\nu(j)) = \emptyset$. In other words, each channel has a uniquely specified component as its source (including the environment as a source). We denote the set of K (identifiers for the) nodes of the net by

$$Nodes((\nu, O))$$

We denote the set of all the channels of the net by Chan((ν, O)) specified by the equation

$$Chan((\nu, O)) = O \cup \{c \in In(\nu(k)): k \in K\} \cup \{c \in Out(\nu(k)): k \in K\}$$

The channels in the set

$$\{c \in Out(\nu(k)): k \in K\} \backslash O$$

are called *internal.* Recall that the mapping ν associates with every node its behavior in the form of an I/O-function.

[1] The fact that we use component identifiers gives components their unique identity over the lifetime of a system. One may think of *object identifiers*.

[2] We assume that the input and output arcs of each node labeled by an identifier k ∈ N are determined exactly by In(ν(k)) and Out(ν(k)) respectively.

A data flow net seen from the outside describes an I/O-function. This I/O-function is called the *black box view* of the distributed system that is described by the data flow net. It defines an abstraction of the distributed system that is represented by the data flow net (ν, O) leading to its black box view by mapping it to a component in COM[I, O]. Here I denotes the set of input channels and O denotes the set of output channels of the data flow net. This black box view is represented by the I/O-function $F_{(\nu, O)} \in \mathrm{COM}[I, O]$ specified by the following formula:

$$F_{(\nu, O)}(x) = \{ y|_O : y|_I = x \wedge \forall\, k \in K : y|_{Out(\nu(k))} \in \nu(k)(y|_{In(\nu(k))}) \}$$

Here we use the notation of function restriction. For a function $g: D \rightarrow R$ and a set $T \subseteq D$ we denote by $g|_T: T \rightarrow R$ the restriction of the function g to the domain T.

The formula essentially expresses that the output history of a data flow net is the restriction of a channel evaluation for all the channels of the net that is a fixpoint[1] for all the net equations to the output channels.

4. Dynamic Systems

The dynamics of information processing systems is not so easy to grasp. On one hand universal programmable computer systems show a very dynamic behavior by definition. They gain their flexibility by the fact that they can be programmed to compute any computable task. However, this form of dynamics is rather specific: it requires human interaction by the programmer. In contrast to this we are interested in this paper in dynamic systems with a dynamics that is part of the programmed system behavior.

A crucial question here is to find the appropriate level of abstraction of computing systems to study their dynamics. On the machine level, where only bits and bytes are processed and even the difference between data and instructions disappears it is quite difficult to capture the notion of dynamics of systems. At this level computers process bit streams. Only at higher levels of abstractions dynamics becomes explicit. On very high levels of abstraction, however, in many cases the dynamics may no longer be explicitly visible.

Another crucial issue for an information processing system and its description is the balance between statics and dynamics. In a world without types, for instance, any behavior can be encoded (see π-calculus [Milner 91]). Types introduce restrictions on the system behavior and this way restrict the dynamics of systems. Obviously, it is crucial to find the right balance between static aspects including types and dynamic aspects. In system development, we want to associate a number of properties, structure, and views with a system model. Due to the dynamics of a system, some of these views may change.

A distributed, interactive system is called *dynamic*, if it changes its set of components or channels, its distribution structure, its topology and/or its channel connection structure during its lifetime. This means that it may change step by step

- its set of existing components,
- the locations of its components,
- its set of communication links and its interconnection structure (internal and external channels).

[1] According to the fact that we consider only time-guarded I/O-functions it can be shown that if the I/O-function is deterministic, there is always a fixpoint and the fixpoint is unique. Due to time guardedness, the recursive equations are sufficient to characterize this fixpoint and the idea of a least fixpoint is not needed.

Along these lines we can even speak in the case of black box views of dynamic systems and of dynamic components, if components change their *syntactic interface* over their lifetime. Of course, state transition machines where the set of components and channels is a part of their state easily model dynamic systems. In such models, however, the distribution structure is rather implicit. Moreover, it is difficult to arrive at modular system models that way with clean and simple composition operators. We are, however, interested in models which present and deal with the distribution structure more explicitly.

4.1 Dynamic Nets and Mobility

A dynamic net typically changes its set of components and its channels during its lifetime. Speaking in more general terms, a dynamic system changes its component structure (its architecture) over its lifetime. In principle, these changes are not very difficult to model. If we model a system by a state transition machine we may describe the connection network as a part of the state. By state transitions the state may change and so may the network. This way we may describe state transition steps with radical changes of the system structure. However, for most applications we are not interested in radical changes of the network structure within one step. Rather we are interested in very specific, small, relatively local changes where in one step most of the net structure remains unchanged and only

- one component is added or deleted, or
- one channel is added to or deleted from a component or its source and/or target is changed.

This leads to the idea of evolving networks along the lines of the π-calculus (see [Milner 91], [Milner et al. 92]) or the ambient calculus (see [Cardelli 95]) where the steps of changes are captured by rewriting rules. In fact, in π-calculus the meaning and behavior of dynamic networks is specified in a purely operational way, which does not lead to a clear notion of an interface nor to a denotational model of a dynamic system. We are interested in the following in a denotational model and its modularity as a basis of specification and design techniques.

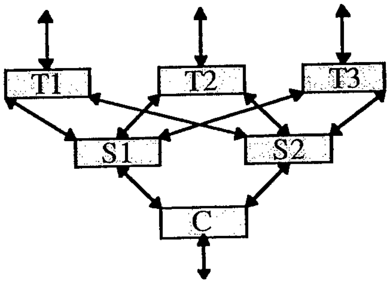


Fig. 2 Static Network of Central C, Stations S1, S2 and Mobile Phones T1, T2, T3

4.1.1 Dynamics of Structure, Interface, and Behavior

We start with an example to illustrate the applications, notions, and goals of mobility and dynamics.

Example: Mobile Telephone
Let us discuss the different options to deal with dynamic systems by a rather simple, well-known example namely that of a mobile telephone system comprising two switching stations, one central telephone exchange, and up to three mobile phones.

As shown in Fig. 2, the system consists of a network which contains as components a telephone central C, two switching stations S1 and S2 and three mobile phones T1, T2, and T3.

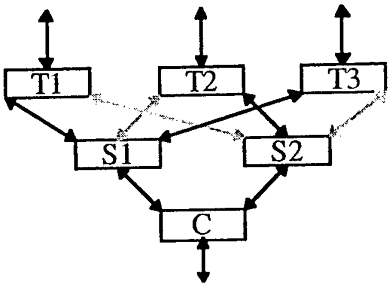


Fig. 3 Dynamic Network with Three Active Mobile Phones (Inactive Channels in Grey)

In this network the component C is static (its set of active channels never changes) as long as both stations are always active while the components S1, S2, T1, T2, and T3 are dynamic since their set of active channels may change.

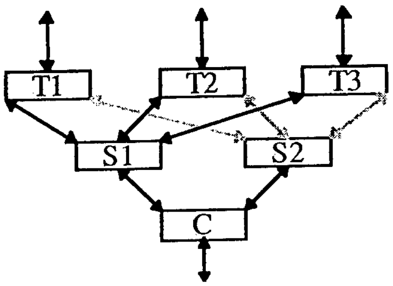


Fig. 4 Dynamic Network with Three Active Mobile Phones and Changed Connection

We may model the case that mobile phones exchange their stations as illustrated by the Fig. 2 to 5.

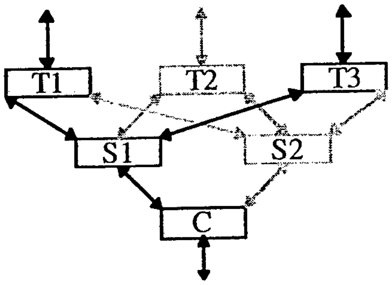


Fig. 5 Dynamic Network with Two Active Phones and One Active Switching Station

Fig. 2 gives the static network of all possible components and all possible connections. These are all the connections and components that may exist over the lifetime of the system. Fig. 3 shows a system configuration where each phone is active and connected to exactly one station. Fig. 4 shows the result of changing the connection for telephone T2 from station S2 to station S1. Fig. 5 shows a configuration where telephone T2 and station S2 are inactive. In this case the central C is a dynamic component, too.

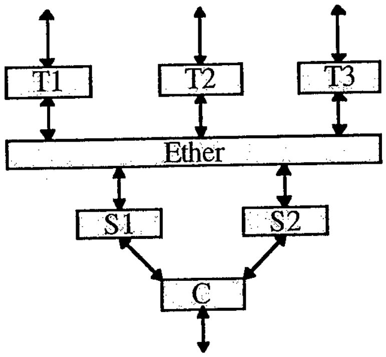


Fig. 6 Dynamic Network with an Ether Connection

Fig. 6 shows a solution where a transmission component called Ether is introduced that acts as a communication bus. In this case, the system structure is static and no dynamic behavior occurs explicitly since the channel and message switching is done by the component Ether and therefore the net appears static.

Note that there are two options to give a behavior to the component Ether. Either the component Ether is a *merge* component (also called a *multiplexer*) that merges all its input message streams and forwards them to all its output channels (broadcasting) or it is a *switch* that connects certain inputs with certain outputs or it is a combination of a filter and a merger. As a special case Ether can be described as a switching network that has a state indicating exactly which channels are connected. This state reflects the net structure as it is shown in Fig. 3 to 5 explicitly. □

In a network of the characteristics introduced in the example above the structure of the system, the set of existing components and connections of the components may change during the lifetime of the system. In fact, the example shows only two essential aspects of dynamics, the dynamics of the interface (in Fig. 5 the switching station S1 has only two (active) channel connections to mobile phones while in Fig. 4 it has three) and the dynamics of the connections and sets of components of a system. In fact, we are also interested in components that change their functionalities by offering modified or additional services.

To model the dynamics of a system we can describe a sequence of states of a system by snapshots. In each snapshot the system distribution and communication connection structure may change. We believe that it is important to keep track of the individual objects (components) during the lifetime of dynamic systems. This is achieved by a particular notion of *identity* for each of the components. This identity is captured easily with the help of unique component identifiers such as in object orientation or in the Internet by the IP addresses. In dynamic systems we are interested to keep track of the individual components. If we compare two snapshots of the system showing the subnet of active components and channels we obtain two different nets, in general. Which of the components of these nets represent the same computing entity can be determined only by the identifiers associated with the components.

4.1.2 Mobility

Interesting aspects in dynamic systems that we did not mention explicitly in our example so far are the individual steps by which the number of components changes. Examples are the melting of components (making one component from several ones), the cloning of components (creating a second copy of a component), or their split (dividing a component into two). These patterns of system behaviors can be seen as special cases of system dynamics that are achieved by naming conventions and/or hierarchical networks. We are interested to treat, in addition to the dynamics of systems, a hierarchical partitioning of networks. We explain this idea again by an example showing the dynamic reconfiguration of systems.

Example: Reconfiguration
We give a very simple syntactic example for reconfiguration. A reconfiguration collects and encapsulates different sets of components into locations. Each of the locations contains a set of components. These sets are forming again components. As a result we get partitions of a system that may change dynamically.

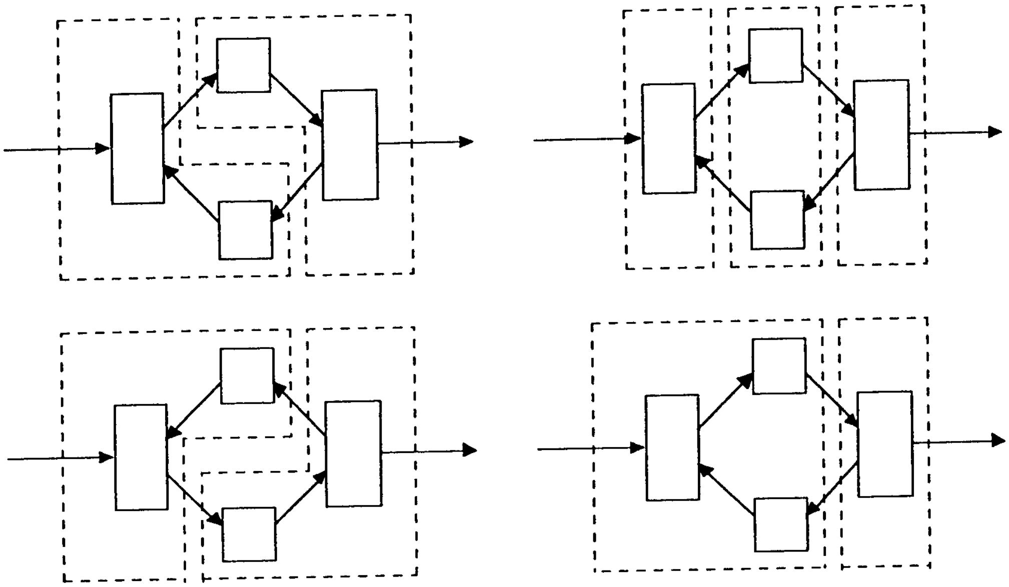


Fig. 7 Four Examples of Configuration of a Simple Net

Fig. 7 shows four different configurations of a simple net. Each corresponds to a particular partition of the set of components. We could also study configurations with nondisjoint sets of overlapping subsystems. We can think about steps of reconfiguration of a system where such changes appear. ❑

The idea of partitions gives us, in particular, one handle to speak about component mobility. By partitions where the elements of the partitions are identified by identifiers called locations we can speak about system state changes where one component moves from one partition to the other.

Configurations define system topologies and their neighborhoods for systems. We may ask in such a system model if two components are in the same set of the partition ("are in the same environment"). This can be used to allow, for instance, additional ways of interactions for components that are in the same set of the partition. Those aspects are modeled in the *ambient calculus* (see [Cardelli 95]).

4.2 A Formal Model for Dynamic Systems

To keep our discussion simple we restrict our discussion to deterministic components and systems in the following. An extension to nondeterministic systems is possible along the lines of Focus [Broy 98], too.

Let I be throughout this section a (possibly infinite) set of input channels and O be a (possibly infinite) set of output channels.

4.2.1 Dynamic Streams

A dynamic stream represents the history of a channel that may become active and inactive several times through its lifetime. Given a type M (which in our setting is simply a set of data elements used as messages) we define the set of dynamic streams over the set M by the set of streams over $M \cup \{@, ©\}$ as follows: s is a dynamic stream over M and we write $s \in M^@$ if the following formula holds:

$$\begin{aligned} \forall t \in \mathbb{N}\backslash\{0\}: \quad & (\neg\alpha(s).t \Rightarrow s.t = \langle\rangle \vee s.t \in \{@\} \times M^*) \\ \wedge \quad & (\alpha(s).t \Rightarrow s.t \in M^* \vee s.t \in M^* \times \{©\}) \end{aligned}$$

where the function

$$\alpha : ((M \cup \{@, ©\})^*)^\infty \to (\mathbb{N} \to \mathbb{B})$$

is an auxiliary function with the meaning α(s).t expresses that the stream s is active at the beginning of time interval t.α is specified by the equations

$$\alpha(s).0 = \text{false}$$

$$\alpha(s).t+1 = (\alpha(s).t \wedge \neg© \in s.t) \vee (\neg\alpha(s).t \wedge @ = \text{ft}(s.t+1))$$

This definition essential defines the patterns of activation and deactivation signals and thus expresses the rules of activating and deactivating a channel. Note that a channel can either be activated and deactivated in one time interval of the system.

Streams on dynamic channels may carry activation and deactivation messages. Therefore the type of a channel indicates whether the channel is dynamic. Here we assume for simplicity that the channels are active or inactive always during complete time intervals. So at the beginning of each time interval (say at time t) we have a set of active channels that stay active over the entire time interval and the other channels stay inactive throughout the entire time interval[1]. For simplicity, all dynamic channels are inactive at time 0.

We introduce a type function for sets of channels C:

$$\text{type}: C \to T$$

where T is the set of all types each type in T is a set. By

$$\vec{C}$$

we denote the set of all channel valuations. A channel valuation $x \in \vec{C}$ is a mapping

$$x: C \to (M^*)^\infty$$

where $M = \cup \{t: t \in T\}$ is the universe of all messages. We require for a channel valuation that

$$x.c \in (t^*)^\infty$$

if type(c) = t . Furthermore, if we have $\{@, ®\} \subseteq \text{type}(c)$ then we require that $x.c \in M^@$ that means that x.c is a dynamic stream.

[1] Note that only due to our model of time we speak so easily about the "state" of the system w.r.t. the activity of channels.

4.2.2 Dynamic Components

A *dynamic component* (deterministic behavior) with the syntactic interface (I, O) corresponds to an I/O-function

$$F: \vec{I} \to \vec{O}$$

where some of the channels in $I \cup O$ are dynamic. The function F models the input/output behavior of the component. On this basis we are able to define a function

$$\alpha: \{x{\downarrow}t: t \in \mathbb{N} \wedge x \in I\} \to \wp(I \cup O)$$

The function α yields for every initial segment $x{\downarrow}t$ of an input history x at time t a snapshot that represents the set of active channels. Only if a channel is active communication may take place along it. The only exceptions are the activation messages (see above). The set $\alpha(x{\downarrow}0)$ yields the set of channels that are active initially, which are the static channels.

The set of dynamic components is denoted by DCOM. The set of dynamic components with the syntactic interface (I, O) is denoted by DCOM [I, O]

Note that we did not make any assumptions about the cardinality of the channel sets for a component. Of course, the set of dynamic channels of a component can be infinite. Whether we are interested in and allow for components with an infinite number of active channels is another question. Certainly, for particular applications components with an unbounded set of active channels are of major interest.

4.2.3 Dynamic Nets

Let DCOM be the set of dynamic components. We work with a function that assigns behaviors to component identifiers. A *dynamic net* consists of a (possibly infinite) set of nodes K (identifiers for components) and a mapping

$$\nu: K \to \text{DCOM}$$

where DCOM is the set of dynamic components. Let I be the set of input channels of the net. To model activation and deactivation of the components in the net, we define a mapping

$$\beta: \{x{\downarrow}k: k \in \mathbb{N} \wedge x \in \vec{I}\} \to \wp(K)$$

This mapping indicates which components of the net are active at a given time. The channels that are active in the network are the active channels of the active components.

For simplicity we assume that a component is inactive, if and only if all its channels are inactive. We define the activity function ß as follows:

$$\begin{aligned}\beta(x).t = \{k \in K: \exists\, y \in \vec{C}: \quad & y|_I = x \\ \wedge\ & \forall\, k \in K: y|_{Out(\nu(k))} \in \nu(k)(y|_{In(\nu(k))}) \\ \wedge\ & (\exists\, c \in In(\nu(k)) \cup Out(\nu(k)): \ \alpha(y.c{\downarrow}t)\ )\}\end{aligned}$$

At a first glance for some readers it may look strange to work in our system model with a set consisting of all the components and of all the channels that might become active during the lifetime of a system. However, this is not so difficult and unusual as it may seem at a first glance. In object-oriented systems the set of object identifiers determines the set of potentially active objects. And the syntactic structure of the methods (where in the body of a method a method of another class is called) determines together with the links between the objects the possible connection structure. According to this model at each time there exists a uniquely defined network of active components and channels. Moreover, we may assume an infinite set of object identifiers for each class. Therefore each class represents an infinite set of components most of which being inactive in a particular state (time) of the system.

Also in other systems it is quite common that the set of potentially active components is determined (and bounded) by a set such as the set of potentially active components. An example is the Internet with its finite number of IP-addresses.

5. Conclusions

It is the main goal of this paper to demonstrate that the methods, notations, and concepts used in practice for the modeling and description of digital systems including dynamics can well be scientifically based on the more foundational and theoretical work created so far in computing science. This way we obtain a mathematical basis for powerful system modeling languages as well as software and system engineering methods.

The benefits of such a mathematical foundation are quite obvious. In particular, we obtain this way:

- A deeper understanding of the methods leading to helpful "Gedankenmodelle" based on the mathematical models.
- Better description techniques for dynamic systems.
- The conceptual consistency of description and development methods for developing dynamic and mobile systems.
- The mathematical basis that can help as a guideline for the definition of development methods for dynamic and mobile systems.
- Advanced tool support for specification as well as consistency checking, prototype generation, simulation, and verification of dynamic systems with a firm basis.

Apart from these more direct benefits of a mathematical foundation of software engineering methods, a scientific foundation is badly needed as a step toward a more systematic study of methods for dealing with dynamic and mobile systems. Only if we manage to develop general common criteria to compare the expressive power and quality of software engineering methods we will be able to free our discipline from dogmatic view points and marketing-based judgments and thus prepare the ground for scientifically justified and practically tractable systems and software engineering methods.

Acknowledgment

The thoughts presented above have benefited greatly from discussions within the SysLab team and the Forsoft I Project Al research group.

References

[Broy 95]
M. Broy: Advanced Component Interface Specification. In: Takayasu Ito, Akinori Yonezawa (Eds.). Theory and Practice of Parallel Programming, International Workshop TPPP'94, Sendai, Japan, November 7-9, 1994, Proceedings, Lecture Notes in Computer Science 907, Springer 1995

[Broy 98]
M. Broy: Compositional Refinement of Interactive Systems Modelled by Relations. In: W.-P. de Roever, H. Langmaack, A. Pnueli (eds.): Compositionality: The Significant Difference. LNCS State of the Art Survey, Lecture Notes in Computer Science 1536, 1998, 130-149

[Cardelli 95]
R. Cardelli: A Language with Distributed Scope. ACM Trans. Comput. Syst. 8, 1 (Jan.), 27-59. Also appeared in POPL 95.

[Milner 91]]
R. Milner: The polyadic π-calculus: A tutorial. Technical Report ECS-LFCS-91-180, University of Edinburgh, 1991.

[Milner et al. 92]]
R. Milner, J. Parrow, D. Walker: A calculus of mobile processes. Part i + ii, Information and Computation, 100:1 (1992) 1-40, 41-77

A formal approach to specification-based black-box testing*

María Victoria Cengarle
Institut für Informatik
Ludwig-Maximilians-Universität München
cengarle@informatik.uni-muenchen.de

Armando Martín Haeberer
Oblog Software S.A.
haeberer@oblog.pt

1 Introduction

This paper introduces an initial account of a formal methodology for specification-based black-box verification testing of software artefacts against their specifications, as well as for validation testing of specifications against the so-called *application concept* [14].

When testing software process artefacts we have three actors. The first is a *posit* we make on the real world, whether it be a *software artefact* reified from a specification or a *software artefact to be*, i.e., the application concept, the *hypothetical posit* we imagined and whose behaviour the specification should capture. In both cases we obtain *evidence* from such a posit—if it is a real software artefact by executing it; if it is a hypothetical posit by producing instances of its hypothetical behaviour. The second actor is the specification, which is a *theory* supposedly explaining the behaviour of the posit. Actually, when testing the relation between a posit and its specification what we test is the 'correctness' of such an explanation. In case the posit is a hypothetical one, we talk about *validation testing*, i.e., the testing activity aims at answering the question 'are we constructing the correct thing?'. In the case the posit is a software artefact, we talk about *verification testing* and the testing activity aims at answering the question 'are we constructing this thing correctly?'. Finally, the third actor is the property we are testing, which is a *hypothesis* we make about the posit and which should be tested using the whole specification as a background theory. In other words, this hypothesis will be true if

*The research reported in this paper was developed with the support of the DAAD (German Academic Exchange Service), the CNPq (Brazilian National Research Council), the Ludwig-Maximilians-Universität München, the EPSRC (Engineering and Physical Sciences Research Council, UK), the Imperial College of Science, Technology and Medicine, London, and PUC-Rio (Pontifícia Universidade Católica do Rio de Janeiro, Brazil).

(and hopefully 'only if') the specification correctly 'explains' the hypothetical posit (validation), or if the software artefact is 'correct' with respect to the specification (verification).

This setting resembles very closely the one of testing of *scientific theories*, i.e., of testing the 'correctness' of the explanation a particular scientific theory supports about certain *phenomena*. As soon as we investigate the relation between the two settings, its resemblance is compelling (see [12, 5]). The specification corresponds to the scientific theory, whilst the posit the specification describes corresponds to the phenomenon the scientific theory explains.

Let us denote by $\mathcal{T}^*$ the specification (or background theory),[1] H the hypothesis under test, and $\underline{E}$ the evidence produced by the posit. The problem of relating the evidence emerging from a phenomenon with the theory explaining it, i.e., the problem of logically explaining how some evidence, which is a piece of observation, can refute or confirm a hypothesis on the basis of a theory[2] explaining such a phenomenon (both stated in a theoretical language), was one of the major issues of the *Philosophy of Science* (*Epistemology*) of the Twentieth Century.

In Fig. **1** the two major strategies to relate theory and evidence are depicted. The most popular one is illustrated on the right-hand side of the figure. There, from the theory $T^* \cup \{H\}$ a prediction E_P about the evidence $\underline{E}$ is derived using the logic underlying both the so-called *theoretical* and *observational* segments of $T^* \cup \{H\}$. (We will succinctly discuss these segments below.) Then, the *experiment* consists in comparing the predicted evidence E_P with the one produced by the posit, i.e., $\underline{E}$. The experiment is *successful* if the prediction holds, *unsuccessful* otherwise. This strategy is called in the epistemological jargon the *hypothetico-deductive* strategy (in the sequel abbreviated to HD); after proposing a hypothesis about the posit, and in the presence of a background theory $\mathcal{T}^* = T^*$, a prediction of an evidence is deduced from the two together and then compared with the actual evidence. As we show below, certain conditions should hold for this strategy to be sound.

The left-hand side of Fig. **1** pictures an alternative strategy, which is also intuitive. This strategy is called the *bootstrap* strategy in the epistemological jargon. Again, the purpose is to test a hypothesis H about a posit on the basis of a background theory, this time $\mathcal{T}^* = (T^* \cup \{H\})^*$.[3] From an evidence $\underline{E}$ produced by the posit and by means of a set of functions $\{f_{\mathcal{T}^*_x}\}$ derived from $\mathcal{T}^*$ (using its underlying logic), obtain a valuation α for the variables $\{x\}$ of H. Then, the *experiment* consists in determining if

[1] Theory presentations are denoted by T, $\mathcal{T}$, etc.; theories obtained from those presentations (i.e., presentations closed under inference) are denoted by T^*, $\mathcal{T}^*$, etc.

[2] In Epistemology literature it is usual to read "on the background of a theory" instead.

[3] By abuse of notation we also write $\mathcal{T}^* = T^* \cup \{H\}$.

the valuation α makes H valid, in which case the experiment is *successful*, otherwise it is *unsuccessful*. As in the case of HD, certain conditions on the derivation of the set $\{f_{\mathcal{T}^*_x}\}$ must hold for the strategy to be sound. The bootstrap strategy (as well as HD) is based on Carnap's ideas; in Glymour's words, *Whenever our evidence is stated in terms narrower than those of our theory, [one of Carnap's ideas] contains a stratagem for making the connection between evidence and theory: use [the background theory] to deduce, from the [evidence], instances of [the hypothesis].*

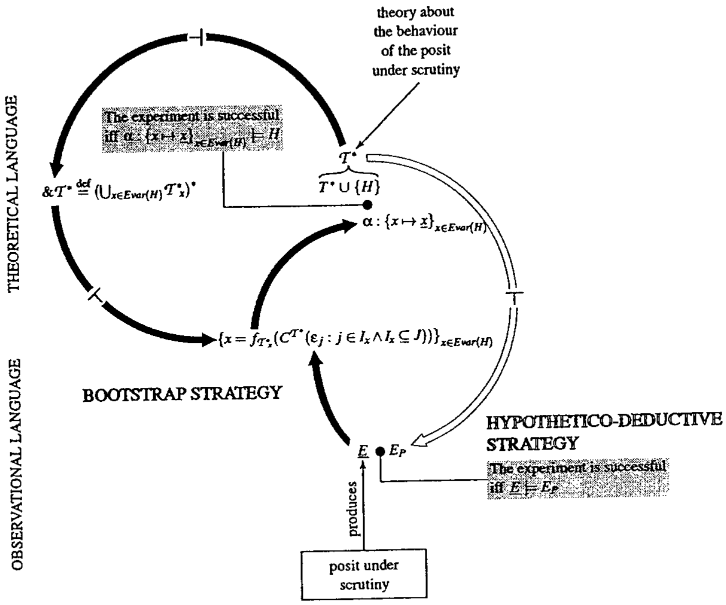

theory about the behaviour of the posit under scrutiny
THEORETICAL LANGUAGE
OBSERVATIONAL LANGUAGE
The experiment is successful iff $\underline{\alpha} : \{x \mapsto \underline{x}\}_{x \in Evar(H)} \models H$
$\mathcal{T}^*$
$\overbrace{\mathcal{T}^* \cup \{H\}}$
$\&\mathcal{T}^* \overset{\text{def}}{=} (\bigcup_{x \in Evar(H)} \mathcal{T}^*_x)^*$
$\alpha : \{x \mapsto \underline{x}\}_{x \in Evar(H)}$
$\{x = f_{\mathcal{T}^*_x}(C^{\mathcal{T}^*}(\varepsilon_j : j \in I_x \wedge I_x \subseteq J))\}_{x \in Evar(H)}$
BOOTSTRAP STRATEGY
HYPOTHETICO-DEDUCTIVE STRATEGY
$\underline{E}$
E_P
The experiment is successful iff $\underline{E} \models E_P$
produces
posit under scrutiny

Figure 1: Testing strategies

According to which kind of testing we are performing, after applying any of the above strategies, *modus tollens* dictates the course to follow. In the case of verification testing of a software artefact against its specification, if the experiment was unsuccessful, then the software artefact must be revised. In the case of validation testing of a specification against a hypothetical posit, if the experiment was unsuccessful, then the specification must be revised. In both cases, if the experiment was successful, the only information we have is exactly that. Transforming this information into a confirmation means either performing infinitely many experiments or introducing some kind of *uniformity* hypothesis about the domain of the evidence enabling a finite partition of it into *uniform subdomains* such that it is sufficient to test one representative of each one of these subdomains. This paper will not deal with this last problem; its purpose is to introduce the bootstrap strategy as an appropriate one for specification-based black-box testing, and to show why HD is woefully inadequate.

In this paper we denote a theory by $\mathcal{T}^*$ to emphasise the fact that a theory is the closure of a (usually finite, modulo axiom schemata) theory presentation (or axiomatisation) $\mathcal{T}^L$ by means of the inference rules of the underlying logic L. Notice that we have here another potentially infinite dimension for testing. Whatever strategy we use, we should test infinitely many hypotheses to cover the whole theory. However, if we have a finite theory presentation (modulo axiom schemata), we can use it instead of the whole theory, i.e., use its axioms as hypotheses.

Notice that evidence E is stated in a different language than theories and hypotheses. For expressing the former, a restricted language denoting observables and with a restricted logic suffices. For instance, the property 'this brick is red' denotes a directly observable fact, i.e., the redness of this particular brick. Moreover, the sentence 'all the bricks I am talking about are red' is a generalised (finite) conjunction of atomic sentences. In other words, the universal quantifier of the logic accompanying the language of observables must be finite, as we cannot observe infinitely many properties. The notion of direct observation can be relaxed to those things observed via 'accepted' instruments, a microscope when observing cells, or an *oracle* when observing software artefact behaviours.

In contrast, the language in which theories and hypotheses are stated must be rich enough to capture concepts, be they observable or not. Moreover, the accompanying logic must provide infinite quantifiers, modalities, etc. For instance, if the domain of the quantifiers were not infinite, a scientific theory would be transformed into an empirical generalisation.

The existence of these two sublanguages and their accompanying logics with different expressive and deductive power, is the root of the problem of testing alluded to above. In Glymour's words [11],

[...] how can evidence stated in one language confirm hypotheses stated in a language that outstrips the first? The hypotheses of the broader language cannot be confirmed by their instances, for the evidence, if framed in the narrower tongue, provides none. Consistency with the evidence is insufficient, for an infinity of incompatible hypotheses may obviously be consistent with the evidence [...].

All these problems were deeply studied by the so-called *logical-empiricist philosophers*, in particular by the members of the *Vienna Circle*. Rudolf Carnap formally introduced the observational-theoretical dichotomy by means of a theory known today as *The Statement View of Scientific Theories* (in short The Statement View) in the 38 years between 1928 [2] and 1966 [3].

In the context of the Statement View, whenever we have an empirically interpreted theory $\mathcal{T}^*$ we have two disjoint subtheories of it, a theoretical one, whose presentation is $\mathcal{T}_T = (\Sigma_T^{\mathcal{T}}, Ax_T^{\mathcal{T}})$, and a purely observational one, whose presentation is $\mathcal{T}_O = (\Sigma_O^{\mathcal{T}}, Ax_O^{\mathcal{T}})$, related by a set of correspondence rules $C^{\mathcal{T}}$, which provides the only empirical interpretation of $\mathcal{T}_T$. Therefore, we have two languages generated by the vocabulary of $\mathcal{T}^*$, the theoretical one (which we will call $\mathcal{L}_T^{\mathcal{T}}$) and the observational one (which we will call $\mathcal{L}_O^{\mathcal{T}}$). Observable facts are stated in $\mathcal{L}_O^{\mathcal{T}}$. We will refer to such observable facts as *evidence*. The requirement that the set of correspondence rules provide the only empirical interpretation of the theoretical subtheory preserves the safeness of observational consequences. That is, some theoretical symbols are given empirical interpretation, by the correspondence rules, in terms of the observational symbols. The theoretical symbols whose meaning is not definitionally given by the correspondence rules are not interpreted further. Their properties are given by relating them with the already interpreted theoretical symbols by means of the deduction mechanisms provided by the underlying logic. This means that there are more theoretical symbols than observational ones. Therefore, it is easy to see that $\mathcal{L}_T^{\mathcal{T}}$ outstrips $\mathcal{L}_O^{\mathcal{T}}$ in expressive power. This difference is due, on the one hand, to the above mentioned 'difference in size' of the corresponding vocabularies, and on the other, to the logic underlying the theoretical and the observational segments of the theory.

Outline In Sect. 2 we present the hypothetico-deductive strategy in detail and discuss its flaws. In Sect. 3 we introduce in detail the bootstrap strategy for deterministic evidence and, in Sect. 4, we apply it to the verification testing of a toy deterministic program. In Sect. 5 the bootstrap strategy is adapted to non-deterministic evidence, whilst in Sect. 6 its application to the verification testing of a toy non-deterministic program is presented.

2 The hypothetico-deductive strategy

Let us recall Fig. 1. In the HD strategy the hypothesis H and the theory T^* are used to derive a prediction E_P, and then some actual evidence $\underline{E}$ is used to determine whether or not the prediction is true, i.e., if $\underline{E} \models E_P$. The hypothesis H is stated in the theoretical language; the background theory T^* is an interpreted theory containing its theoretical part, its observational part, and its correspondence rules; finally, the prediction must obviously be stated in the observational language. In order to derive the prediction from the union of the theory and the hypothesis, the theoretical terms appearing in the latter must have a direct (by means of some correspondence rules) or indirect (by means of inference and some correspondence rules) empirical interpretation.

A HD schema is a triple $\langle T^*, H, E_P \rangle$ where T^*, H, and E_P are as in our scenario above, and the following three conditions hold:

(i) $T^* \cup \{H\}$ is consistent

(ii) $H, T^* \vdash E_P$

(iii) $T^* \nvdash E_P$

Then, as we said before, $\underline{E} \models E_P$ does not refute H w.r.t. T^*, whilst $\underline{E} \not\models E_P$ refutes H w.r.t. T^*.

Let us analyse these conditions. The first one is obvious; if $T^* \cup \{H\}$ were inconsistent, then any prediction could be derived from it. The second condition is the essence of the HD strategy. Finally, the third condition prevents us from affirming that H is tested in the case that T^* suffices for predicting E_P. Notice that we are using T^* instead of $\mathcal{T}^*$: the reason is the necessity of stating condition (iii). Therefore, when using the HD strategy, we should consider $T^* \cup \{H\}$ as the theory explaining (making an appropriate prediction E_P) for $\underline{E}$, but H can only be confirmed w.r.t. some T^*, i.e., w.r.t. a subtheory of $\mathcal{T}^*$.

The outstanding problems of the HD strategy are the following. First, $\underline{E}$ can never refute or confirm (for the precise meaning of confirmation recall the discussion of page 3 in the introduction) w.r.t. $\mathcal{T}^*$ any consequence of $\mathcal{T}^*$ itself; notice, however, that, if H is a consequence of $\mathcal{T}^*$, then it can be confirmed with respect to $H \to \mathcal{T}^*$, i.e., w.r.t. a subtheory of the original theory $\mathcal{T}^*$ such that conjoining it with the hypothesis is logically equivalent to $\mathcal{T}^*$ (see [10]). Second, if E_P is not a tautology, $\underline{E} \models E_P$, and L is any consistent sentence such that $E_P \nvdash L$, then L is confirmed by $\underline{E}$ w.r.t. a true theory (namely $L \to E_P$). Third, if H is confirmed by $\underline{E}$ w.r.t. T^*, then so is $H \wedge K$, where K is any sentence whatsoever that is consistent with H and T^*. (Recall the property known as *reinforcement of the antecedent* of classical propositional logic.) The first difficulty might be tolerated were it not for the other two; together, they make the HD account untenable.

There were different attempts to save the HD strategy. Let us consider, for instance, Merrill's attempt to overcome the third difficulty (see [15]). Additionally to the above listed three conditions, we should corroborate the fact that there do *not* exist sentences K, L, and M such that:

(iv) $\vdash H \leftrightarrow K \wedge L$

(v) $K \nvdash H$

(viii) $L \nvdash H$

(ix) $M \nvdash H$

(viii) $\nvdash M \leftrightarrow L$

(ix) $K, M, T^* \vdash E_P$

(x) $T^* \cup \{K, M\}$ is consistent

which in English means that if H is a conjunction, then no one of its conjuncts L (or any M equivalent to L) suffices for deriving E_P.

Unfortunately, Glymour showed (see [10]) that this addition leads to circular reasoning, i.e., $E_P \vdash H$. The proof can be sketched as follows. Assume that T^*, H, and E_P satisfy (i), (ii), and (iii) above.

Suppose that $\vdash (H \leftrightarrow (T^* \leftarrow E_P))$, then $(T^* \leftarrow E_P) \vdash H$ by the deduction theorem and, given that $E_P \vdash (T^* \leftarrow E_P)$, then $E_P \vdash H$. Suppose now that $\nvdash (H \leftrightarrow (T^* \leftarrow E_P))$, then we let K be $(T^* \leftarrow E_P)$, L be $((T^* \leftarrow E_P) \leftarrow H)$, and M be any tautology. In this case, and using the deduction theorem and modus ponens, as well as reasoning by *reductio ad absurdum*, it can be shown that K, L, and M satisfy (iv)–(x). Thus if E_P is a prediction by means of the hypothetico-deductive strategy, then necessarily $\vdash (H \leftrightarrow (T^* \leftarrow E_P))$ as in the first case above, which means that $E_P \vdash H$.

Moreover, Glymor has proved in [10] that another suggested attempt to save the HD strategy by adding additional constraints does not help. Even a late attempt to lend credibility to hypothetico-deductivism by using relevance logic (see [16]) instead of classical logic was not successful, since relevance logic itself was not yet well accepted.

Because of the failure of the HD strategy, specification-based black-box testing methods based on it (or on rudimentary versions of it), for instance the method proposed in [7, 8, 9] (for its criticism on the basis of this setting see [5]), have inherent and insurmountable problems. This problem led us to the consideration of the bootstrap strategy as an alternative basis for a methodology for specification-based black-box verification testing of software artefacts and for validation testing of specifications against the application concept.

3 The bootstrap strategy for deterministic evidence

We present here the bootstrap testing strategy for the case of theories over existential (in)equational logic EEQ^+ (called simply EEQ if inequalities are not involved), i.e., the logic underlying systems of n (in)equations with m unknowns. For the sake of simplicity, we assume that there is no α-conversion.

Let $\mathfrak{A} = \langle A, \leq \rangle$ be an algebra with a natural order. Let $\mathcal{T}^*$ be a theory, let H be an equation (or inequality), and let both be mutually consistent. Let $E = \{\varepsilon_j : j \in J\}$ be a set of variables, and let $\underline{E} = \{\underline{\varepsilon_j} : \underline{\varepsilon_j} \in A$ and $j \in J\}$ be a set of values for the variables in E (these variables and their values belong to the observational language), which once lifted by the set of correspondence rules $C^{\mathcal{T}^*}$ are consistent with H and $\mathcal{T}^*$.

Bootstrap testing schemata are three place relations $\langle \mathcal{T}^*, H, \underline{E} \rangle$, where $\mathcal{T}^*$ is a theory, H the hypothesis to be tested with respect to this theory, and $\underline{E}$ the evidence which can refute H with respect to $\mathcal{T}^*$. In general, to be considered a bootstrap testing schema, $\langle \mathcal{T}^*, H, \underline{E} \rangle$ must satisfy a set of conditions such as (i) to (iv) below. Of the bootstrap schemata we introduce only one. All of them coincide in satisfying conditions (i), (ii) and (iv) below; the difference between them resides in the requirement stated by each one's condition (iii).

We begin by stating the Schema I for the deterministic case. In order to do so, we need to introduce some concepts and notation. A *subtheory* of a theory $\mathcal{T}^*$ is a theory $\mathcal{T}_1{}^*$ such that $\mathcal{T}^* \vdash \mathcal{T}_1{}^*$; two theories are *equivalent* if each one is a subtheory of the other. A variable is *essential* to an (in)equation K if it occurs in every (in)equation equivalent to K; the set of essential variables of K is denoted by $Evar(K)$. (Recall that we do not have α-conversion.)

The Schema I of bootstrap testing is defined as follows. Given $\langle \mathcal{T}^*, H, \underline{E} \rangle$, for each $x \in Evar(H)$ let $\mathcal{T}^*_x$ be a subtheory of $\mathcal{T}^*$ such that:

(i) $\mathcal{T}^*_x$ determines (the value of) x as a function of a set of variables indexed by I_x,
which is a subset of the evidence $E = \{\varepsilon_j : j \in J\}$,
denoted by $x = f_{\mathcal{T}^*_x}(C^{\mathcal{T}^*}(\varepsilon_j : j \in I_x \wedge I_x \subseteq J))$.[4]

(ii) The set of values for the variables in $Evar(H)$
given by $\underline{x} = f_{\mathcal{T}^*_x}(C^{\mathcal{T}^*}(\underline{\varepsilon_j} : j \in I_x \wedge I_x \subseteq J))$
satisfies H.

(iii) There is no (in)equation K with $Evar(K) \subset Evar(H)$
such that $H, \&\mathcal{T}^* \vdash K$ and $K, \&\mathcal{T}^* \vdash H$.[5]

(iv) For all $x \in Evar(H)$, there is no (in)equation K with $Evar(K) \subset Evar(\mathcal{T}^*_x)$
such that $\vdash H \wedge \mathcal{T}^*_x \leftrightarrow \{K\}$.

If conditions (i) to (iv) above are met, $\underline{E}$ is said to provide a *positive test* of H with respect to $\mathcal{T}^*$.

The motivation for the conditions above is as follows:

Condition (i) The requirement that a value be determined for each quantity occurring essentially in H reflects a common prejudice against theories containing quantities that cannot be determined from the evidence. Given $\langle \mathcal{T}^*, H, \underline{E} \rangle$, when values for the basic quantities occurring in H have not been determined from the evidence $\underline{E}$ using some theory $\mathcal{T}^*$, then $\underline{E}$ and the relevant fragment of $\mathcal{T}^*$ do not of themselves provide reason to believe that those basic quantities are related as H claims them to be.

[4] We denote by $f_{\mathcal{T}^*_x}$ the function (determined by subtheory $\mathcal{T}^*_x$) which assigns a value to the essential variable x as a function of the $\{\varepsilon_j : j \in I_x \wedge I_x \subseteq J\}$ translated by the set of correspondence rules $C^{\mathcal{T}^*}$.

[5] $\&\mathcal{T}^* \overset{\text{def}}{=} (\bigcup_{x \in Evar(H)} \mathcal{T}^*_x)^*$. Notice that if $\mathcal{T}^*_x$ is presented by an axiomatisation $\mathcal{T}_x$, then $\&\mathcal{T}^* \overset{\text{def}}{=} (\bigcup_{x \in Evar(H)} \mathcal{T}_x)^*$. In this latter case we denote by $\&\mathcal{T}$ the axiomatisation $\bigcup_{x \in Evar(H)} \mathcal{T}_x$.

Condition (ii) Obvious.

Condition (iii) Suppose there exists an (in)equation K such that $Evar(K) \subset Evar(H)$, $H, \&\mathcal{T} \vdash K$ and $K, \&\mathcal{T} \vdash H$. Let $y \in Evar(H)$, $y \notin Evar(K)$. This means that y is essential to H, but not to H in conjunction with $\&\mathcal{T}$. In other words, y could take any value, independent of the evidence $\{\underline{\varepsilon_j} : j \in I_y \wedge I_y \subseteq J\}$. Therefore, the evidence $\underline{E} = \{\underline{\varepsilon_j} : \underline{\varepsilon_j} \in A \text{ and } j \in J\}$ and the method $\{x = f_{\mathcal{T}_x}(C^{\mathcal{T}^*}(\varepsilon_j : j \in I_x \wedge I_x \subseteq J))\}_{x \in Evar(H)}$ of computing quantities in $Evar(H)$ from the evidence would fail to test the constraints H imposes on y. Thus, a significant component of what H says would go untested.

Condition (iv) Consider the theory presentation $\mathcal{T} : \{x = y, c = d\}$ and the hypothesis $H : x = y$ with $E = \{x, y, c, d\}$.[6] For simplicity, and because the theoretico-observational distinction is not relevant for this discussion, let us suppose that the correspondence rules are, in this case, identity functions, therefore, we identify observational and theoretical variables. The set $Evar(H)$ is $\{x, y\}$, and a positive test of H w.r.t. $\mathcal{T}$ is any set $\underline{E} = \{\underline{x}, \underline{y}, \underline{c}, \underline{d} \mid \underline{x} = \underline{y}\}$, because, applying conditions (i) to (iv), (i.e., the schema above), $\mathcal{T}_x : x = x$ and $\mathcal{T}_y : y = y$. This means that whatever values c and d take, the hypothesis $H : x = y$ will not be refuted with respect to theory $\mathcal{T}$ provided the evidence satisfies $\underline{x} = \underline{y}$. Notice that a $\mathcal{T}'_x : x = y + (c - d)$ is rejected by condition (iv) because there exists $K : y = y + c - d$ with $Evar(K) = \{y, c, d\}$ included in $Evar(\mathcal{T}'_x) = \{x, y, c, d\}$ such that $\vdash H \wedge \mathcal{T}'_x \leftrightarrow K$. If we eliminate condition (iv) and, therefore, accept $\mathcal{T}'_x : x = y + (c - d)$, then the evidence $\underline{E} = \{\underline{x}, \underline{y}, \underline{c}, \underline{d} \mid \underline{x} = \underline{y} \wedge \underline{c} \neq \underline{d}\}$ will refute $H : x = y$ although $\mathcal{T}$ does not establish any link between variables x and y, on the one hand, and c and d, on the other.

4 Bootstrap testing of a deterministic program

In this section we apply the Schema I of bootstrap testing presented above to an example. First, let us recall that the way of declaring our intention of what is observable and what is not, is by stating appropriate correspondence rules relating the evidence with the theory (that is, observables can be single quantities and not necessarily a whole sort).

Let SP^{EEQ} be the specification (or theory presentation) over the logic EEQ of a program with input $\{x_1, x_2, x_3, x_4\}$ and output x_{10} given at the top of Fig. 2, where Ax^{BA} is the set of axioms of

[6] We denote by $L : \{\varphi_1, \ldots, \varphi_n\}$ a system named L consisting of the formulae $\varphi_1, \ldots, \varphi_n$. This is abbreviated to $L : \varphi$ if the system consists of just one formula.

Boolean algebra. Suppose we want to verify the integrated circuit at the bottom of Fig. **2** against the specification SP^{EEQ}. The evidence is obviously constituted by sets $\underline{E} = \{\underline{\varepsilon_1}, \underline{\varepsilon_2}, \underline{\varepsilon_3}, \underline{\varepsilon_4}, \underline{\varepsilon_5}\}$ of values for variables ε_1, ε_2, ε_3, ε_4, and ε_5.

Assume that the values of ε_1, ε_2, ε_3, ε_4, and ε_5 in CH0106 are related, respectively, to the variables x_1, x_2, x_3, x_4, and x_{10} in SP^{EEQ} as detailed in Fig. **2**. Notice that the values $\underline{\varepsilon_i}$ for ε_i ($i = 1,2,3,4,5$) can be either 0V or 5V (where V stands for *volts*), whilst variables x_1, x_2, x_3, x_4, and x_{10} in SP^{EEQ} can only take values **0** and **1**.

The set C^{SP^*} of correspondence rules is therefore constituted by five rules:

- four correspondence rules $C_1^{SP^*}$, $C_2^{SP^*}$, $C_3^{SP^*}$, and $C_4^{SP^*}$, describing the procedure for introducing into the pins labelled ε_1, ε_2, ε_3, and ε_4 of the integrated circuit CH0106, an appropriate signal (0V or 5V) corresponding to a particular valuation (**0** or **1**) for variables x_1, x_2, x_3, and x_4, as well as
- a correspondence rule $C_5^{SP^*}$ describing the measurement procedure to be applied to the output pin labelled ε_5 of CH0106 for assigning to variable x_{10} its corresponding value (i.e., **0** or **1**).

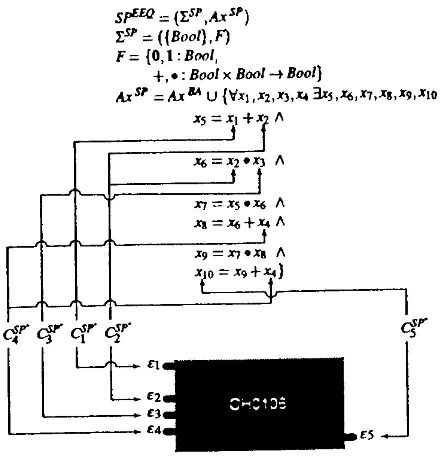

$SP^{EEQ} = (\Sigma^{SP}, Ax^{SP})$
$\Sigma^{SP} = (\{Bool\}, F)$
$F = \{\mathbf{0}, \mathbf{1} : Bool,$
$+, \bullet : Bool \times Bool \to Bool\}$
$Ax^{SP} = Ax^{BA} \cup \{\forall x_1, x_2, x_3, x_4\ \exists x_5, x_6, x_7, x_8, x_9, x_{10} \mid$
$x_5 = x_1 + x_2\ \wedge$
$x_6 = x_2 \bullet x_3\ \wedge$
$x_7 = x_5 \bullet x_6\ \wedge$
$x_8 = x_6 + x_4\ \wedge$
$x_9 = x_7 \bullet x_8\ \wedge$
$x_{10} = x_9 + x_4\}$
$C_4^{SP^*}$
$C_3^{SP^*}$
$C_1^{SP^*}$
$C_2^{SP^*}$
$C_5^{SP^*}$
ε_1
ε_2
ε_3
ε_4
ε_5
CH0106

Figure 2: A testing setting

In order to verify the circuit against the specification, we have to derive a set of experiments. Each one of these experiments tests the circuit against a particular hypothesis H on the basis of the (theory generated by the) specification (theory presentation) SP^{EEQ}. The set of chosen hypotheses must cover SP^*. Therefore, a natural choice for the hypotheses are the axioms of the presentation SP^{EEQ}. Let us test, for instance, $H : x_7 = x_5 \bullet x_6$.

At this point let us emphasise that, as is the case in this example, we can do this, *even when the essential variables of the hypothesis are not related to any symbol of the observational vocabulary by any correspondence rule.*

In the general case, what the test can do is to refute our theory or, at best, not to refute it. In this particular case, since the observable variables range over the Booleans, i.e., the set $\{\mathbf{0}, \mathbf{1}\}$, there is a finite set of possibilities for the evidence, i.e., each one of the combinations of valuations of ε_1, ε_2, ε_3, ε_4, and ε_5 with the set $\{0V, 5V\}$. Therefore, we can exhaustively test H with respect to SP^{EEQ}. Thus, in this particular case, we can confirm H.

Given that $Evar(H) = \{x_5, x_6, x_7\}$, three subtheories $SP^*_{x_5}$, $SP^*_{x_6}$, and $SP^*_{x_7}$ of SP^* should be derived for determining the values for the variables in $Evar(H)$ as functions of $C^{SP^*}(\underline{E})$, where $C^{SP^*}(\underline{E})$ is the evidence $\underline{E}$ 'seen' through the set of correspondence rules C^{SP^*}.

First attempt

For instance, we can present the subtheories as follows:

$$SP_{x_5} : x_5 = x_1 + x_2$$

$$SP_{x_6} : x_6 = x_2 \bullet x_3$$

$$SP_{x_7} : x_7 = (x_1 + x_2) \bullet x_2 \bullet x_3$$

Therefore, we have

$$H, \&SP : \begin{cases} x_7 = x_5 \bullet x_6 \\ x_5 = x_1 + x_2 \\ x_6 = x_2 \bullet x_3 \\ x_7 = (x_1 + x_2) \bullet x_2 \bullet x_3 \end{cases}$$

Notice that $\&SP \vdash H$, and therefore condition (iii) is violated because there exists K, namely $K : x_7 = x_7$, with $Evar(K) = \{x_7\} \subset Evar(H) = \{x_5, x_6, x_7\}$ and both $H, \&SP \vdash K$ and $K, \&SP \vdash H$ (in fact, K could be any tautology with fewer variables than H).

```
program P1
var w1, w2, w3, w4:Bool
function +BBH(x,y:Bool):HBool
function +HBH(x:HBool;y:Bool):HBool
function +HBB(x:HBool;y:Bool):Bool
function •HHH(x:HBool;y:HBool):HBool
function •BBH(x:Bool;y:Bool):HBool
function a(x,y,z,t:Bool):HBool
return x +BBH y
function b(x,y,z,t:Bool):HBool
return y •BBH z
function c(x,y,z,t:Bool):HBool
return a(x,y,z,t) •HHH b(x,y,z,t)
function d(x,y,z,t:Bool):HBool
return b(x,y,z,t) +HBH t
function e(x,y,z,t:Bool):HBool
return c(x,y,z,t) •HHH d(x,y,z,t)
function z(x,y,z,t:Bool):Bool
return e(x,y,z,t) +HBB t
begin
    input(w1, w2, w3, w4)
    output(z(w1,w2,w3,w4))
end
```

Figure 3: A possible implementation

What has violated condition (iii)? We have used a misleading way of calculating a value for the essential variable x_7, i.e., a biased value calculated by using only the 'input evidence' $\underline{E} \setminus \{\underline{\varepsilon_5}\}$ instead of a value calculated involving the measurement $C^{SP^*}_{1 \le i \le 5}(\underline{E})$ of the whole 'input/output' evidence $\underline{E}$. What was wrong in this first attempt? The choice of $\&SP^*$.

Second attempt

So, we must obtain a more appropriate set of subtheories $SP^*_{x_i}$. Notice that, from SP^{EEQ}, we can derive

$$\begin{aligned}
x_{10} &= x_9 + x_4 \\
&= (x_7 \bullet x_8) + x_4 \\
&= (x_7 \bullet (x_6 + x_4)) + x_4 \\
&= (x_7 \bullet ((x_2 \bullet x_3) + x_4)) + x_4 \\
&= (x_7 + x_4) \bullet ((x_2 \bullet x_3) + x_4 + x_4) \\
&= (x_7 + x_4) \bullet ((x_2 \bullet x_3) + x_4) \\
&= (x_7 \bullet x_2 \bullet x_3) + x_4
\end{aligned}$$

multiplying both sides by $\overline{x_4}$,

$$\begin{aligned}
x_{10} \bullet \overline{x_4} &= (x_7 \bullet x_2 \bullet x_3 \bullet \overline{x_4}) + (\overline{x_4} \bullet \overline{x_4}) \\
&= x_7 \bullet x_2 \bullet x_3 \bullet \overline{x_4}
\end{aligned}$$

which functionally determines a value for x_7 iff $x_2 \bullet x_3 \bullet \overline{x_4} = \mathbf{1}$, i.e., iff $x_2 = \mathbf{1} = x_3$ and $x_4 = \mathbf{0}$. Therefore, we present

$$SP_{x_7} : x_{10} = x_7$$

From SP_{x_7}, H, and SP^{EEQ}, one can derive

$$\begin{aligned}
x_{10} &= x_5 \bullet x_6 \\
&= (x_1 + x_2) \bullet x_2 \bullet x_3 \\
&= x_2 \bullet x_3
\end{aligned}$$

which is the so-called *representative* of H.

Notice that

$$H, \& SP : \begin{cases} x_7 = x_5 \bullet x_6 \\ x_5 = x_1 + x_2 \\ x_6 = x_2 \bullet x_3 \\ x_{10} = x_7 \end{cases}$$

and there is no K that could violate condition (iii).

Hence, given the restriction $\underline{x_2} = \mathbf{1} = \underline{x_3}$ and $\underline{x_4} = \mathbf{0}$ to the valuation of variables x_1, x_2, x_3, x_4, and x_{10} imposed by the subtheories $SP^*_{x_5}$, $SP^*_{x_6}$, and $SP^*_{x_7}$, the set of test cases is reduced to the two instances obtained by setting x_1 to $\mathbf{0}$ and observing if the value of x_{10} is $\mathbf{0}$ or $\mathbf{1}$, and repeating the same observation after setting x_1 to $\mathbf{1}$. In order to do this, we should use the procedures described in the correspondence rules $C_1^{SP^*}$, $C_2^{SP^*}$, $C_3^{SP^*}$, and $C_4^{SP^*}$ for applying the correct signals to the input pins ε_1,

ε_2, ε_3, and ε_4; and the procedure described in $C_5^{SP^*}$ for measuring the output value in the pin ε_5. Notice that the evidence will provide a positive test of H with respect to SP^* only in the cases in which the value of x_{10} is **1**, i.e., when ε_5 is measured to be $5V$.

A possible implementation of SP^* is the functional program `P1` in Fig. **3**. Notice, however, that 'an intelligent' programmer or an optimising transformation system (for instance) could have produced the imperative program `P2` in Fig. **4**. In this alternative implementation there are no Boolean functions at all; nevertheless hypothesis $H : x_7 = x_5 \bullet x_6$ can still be tested because SP^{EEQ} also explains the behaviour of `P2`. This example shows the power of the bootstrap strategy in performing black-box testing taking into account only the specification structure, the property under test, and the input/output relation of the program implementing the specification. In [5] we show that `P2` would be rejected by the approach of [9], simply because the `P2` does not implement every axiom of the specification.

```
program P2
var w1,w2,w3,w4:Bool;count:Int
begin
   input(w1,w2,w3,w4)
   if w4 then output(true)
   else count:=0
      if w2 then count:=count+1 endif
      if w3 then count:=count+1 endif
      if count>1 then output(true)
      else output(false)
      endif
   endif
end
```

Figure 4: An alternative implementation

5 The bootstrap strategy for non-deterministic evidence

Bootstrap schemata for deterministic evidence provide means for determining whether or not some evidence provides a positive test of a hypothesis with respect to a theory, where a hypothesis is either an equation or an inequality. However, even the latter option of the hypothesis being an inequality does not fully account for the application of bootstrap testing to non-deterministic sets of evidence (as for example the one generated by a non-deterministic program), since the subtheories $\mathcal{T}_x^*$ must functionally determine a valuation for $x \in Evar(H)$. (In the simplest case, they explicitly describe functions.) For making bootstrap testing fully applicable to the case of non-deterministic sets of evidence, we need to generalise the setting by allowing the subtheories to (non-vacuously) include inequalities so as to allow the variables to become set valued. (In the discussion below, given a function $f : D \to I$ and given $E \subseteq D$, we let $f(E)$ denote the set $\{f(d) : d \in E\}$.)

Now suppose that, in a setting such as that for schemata for deterministic evidence above, for each $j \in J$, the evidence ε_j takes its values from a set $\underline{\Delta}_j$. Then, for each $x \in Evar(H)$ we have five possibilities, namely:

1. If $x = f_{\mathcal{T}^*_x}(C^{\mathcal{T}^*}(\varepsilon_j : j \in I_x \wedge I_x \subseteq J))$,

   then we let $\underline{x} = f_{\mathcal{T}^*_x}(C^{\mathcal{T}^*}(\underline{\Delta_j} : j \in I_x \wedge I_x \subseteq J))$.

2. If $x > f_{\mathcal{T}^*_x}(C^{\mathcal{T}^*}(\varepsilon_j : j \in I_x \wedge I_x \subseteq J))$,

   then we let $\underline{x} = \{a : \text{if } b \in f_{\mathcal{T}^*_x}(C^{\mathcal{T}^*}(\underline{\Delta_j} : j \in I_x \wedge I_x \subseteq J)), \text{ then } a > b\}$.

   (An alternative is to make $\underline{x} = \{a : \text{ there exists } b \in f_{\mathcal{T}^*_x}(C^{\mathcal{T}^*}(\underline{\Delta_j} : j \in I_x \wedge I_x \subseteq J)) \text{ s.t. } a > b\}$.)

3. If $x < f_{\mathcal{T}^*_x}(C^{\mathcal{T}^*}(\varepsilon_j : j \in I_x \wedge I_x \subseteq J))$,

   then we valuate x analogously to the preceding case.

4. If $x \leq f_{\mathcal{T}^*_x}(C^{\mathcal{T}^*}(\varepsilon_j : j \in I_x \wedge I_x \subseteq J))$,

   then the value of $\underline{x}$ is calculated as the union of the set values given by cases 1 and 3.

   In case $x \geq f_{\mathcal{T}^*_x}(C^{\mathcal{T}^*}(\varepsilon_j : j \in I_x \wedge I_x \subseteq J))$, we proceed analogously.

5. If the value of $x \in Evar(H)$ is determined by a collection of inequalities, then the (set-value) of x is the intersection of the sets given by those inequalities separately.

Now, evidence $\underline{E} = \{\underline{\Delta_j} : j \in J\}$ provides a positive test of a hypothesis H with respect to a theory $\mathcal{T}^*$ according to the bootstrap schemata for deterministic evidence, if on the one hand, for each $x \in Evar(H)$ there exists a value in $\underline{x}$ such that these values satisfy H in the usual way, and on the other hand, the other conditions of the schema in use are met.

The reader might be disturbed by the requirement that a single value of $\underline{x}$ for each $x \in Evar(H)$ satisfying H suffices. This seems to mean that a non-deterministic program is considered correct when at least the value produced in one of its executions satisfies H. What about the values $\underline{x}$ produced in other executions not satisfying H, should they be accepted? However, we must recall that the definition of the bootstrap schemata requires that $\underline{E}$ be consistent with H and $\mathcal{T}^*$, and therefore, both pre- and postconditions of the program must be satisfied for the evidence $\underline{E}$ to be able to provide a positive test of H w.r.t. $\mathcal{T}$. This requirement on the evidence $\underline{E}$ stands for an universal quantification overriding the troublesome existential one above for all input/output (observable) variables. Therefore, the weak existential condition above applies only to internal (non-observable) variables.

6 Bootstrap testing of a non-deterministic program

$$
\begin{array}{l}
CONV^{EEQ^+} = (\Sigma^{CONV}, Ax^{CONV}) \\
\Sigma^{CONV} = (\{\mathbb{Q}\}, F) \\
F = \{+,-,\cdot,/ : \mathbb{Q}\times\mathbb{Q}\rightarrow\mathbb{Q}\} \\
Ax^{CONV} = \{\forall m_1,n_1,m_2,n_2\,\exists m_3,n_3,x,y \mid \begin{array}[t]{ll}
0 < m_1 < m_2 & \wedge \\
0 < n_1 < n_3 \leq \frac{m_2n_1-m_1n_2+m_3(n_2-n_1)}{m_2-m_1} & \wedge \\
n_2 < n_1 & \wedge \\
m_3 < 0 & \wedge \\
y \leq m_1x+n_1 & \wedge \\
y \geq m_2x+n_2 & \wedge \\
y \leq m_3x+n_3 & \wedge \\
x > 0 & \wedge \\
y > 0 & \}
\end{array}
\end{array}
$$

Figure 5: The specification $CONV^{EEQ^+}$

Let us now apply the variant of Schema I for theories containing inequalities and set-valued variables to the verification of a non-deterministic program.

Consider the specification $CONV^{EEQ^+}$ over the logic EEQ^+ given in Fig. 5. (We assume that the universally quantified variables are sorted by the correspondence rules in such a way that their constraints, i.e., those conditions involving only those variables and perhaps constants, are satisfied.)

A geometrical interpretation of this specification is the one depicted in Fig. 6, where the surface filled with the parallel vertical line pattern represents the convex polygon defined by the inequalities $y \leq m_1x+n_1$, $y \geq m_2x+n_2$, $y \leq m_3x+n_3$, $x > 0$, and $y > 0$. (That is, the polygon enclosed by the lines $L_1 : y = m_1x+n_1$, $L_2 : y = m_2x+n_2$, $L_3 : y = m_3x+n_3$, and the coordinate axes.) Notice that the convex polygon in Fig. 6 satisfies the conditions $0 < m_1 < m_2$ and $m_3 < 0$ in $CONV^{EEQ^+}$. Notice as well that the point $\langle x_0, y_0\rangle$ exists because $m_1 \neq m_2$, and that therefore, $k = \frac{m_2n_1-m_1n_2+m_3(n_2-n_1)}{m_2-m_1}$ also exists. Finally, notice that the convex polygon in Fig. 6 also satisfies conditions $0 < n_1 < n_3 \leq k$ and $n_2 < n_1$ in $CONV^{EEQ^+}$.

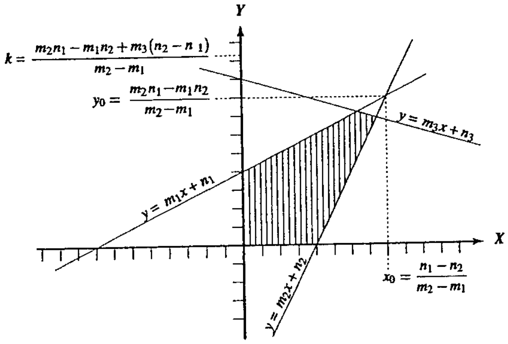

Y
X
$k = \frac{m_2n_1 - m_1n_2 + m_3(n_2 - n_1)}{m_2 - m_1}$
$y_0 = \frac{m_2n_1 - m_1n_2}{m_2 - m_1}$
$y = m_3x + n_3$
$y = m_1x + n_1$
$y = m_2x + n_2$
$x_0 = \frac{n_1 - n_2}{m_2 - m_1}$

Figure 6: A geometrical interpretation of $CONV^*$

As is the case with any specification, $CONV^{EEQ^+}$ can specify many programs, in particular the non-deterministic program `Pnondet` whose input/output diagram is the one depicted in Fig. 7. We will consider that the correspondence rules are identity functions, therefore for the sake of simplicity we will use the set $\{m_1, n_1, m_2, n_2, x, y\}$ as the evidence, instead of using the actual evidence $E = \{\varepsilon_1, \varepsilon_2, \varepsilon_3, \varepsilon_4, \varepsilon_5, \varepsilon_6\}$ and the set of correspondence rules. Here, we assume that the universally quantified variables are sorted by the correspondence rules in such a way that constraints imposed on them by $CONV^{EEQ^+}$ are satisfied. In other words the data $\{\underline{m_1},\underline{n_1},\underline{m_2},\underline{n_2}\}$ with which program Pnondet is fed is such that $0 < \underline{m_1} < \underline{m_2} \wedge \underline{n_2} < \underline{n_1} \wedge 0 < \underline{n_1}$.

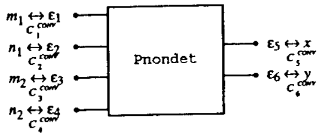

$m_1 \underset{C_1^{conv}}{\leftrightarrow} \varepsilon_1$
$n_1 \underset{C_2^{conv}}{\leftrightarrow} \varepsilon_2$
$m_2 \underset{C_3^{conv}}{\leftrightarrow} \varepsilon_3$
$n_2 \underset{C_4^{conv}}{\leftrightarrow} \varepsilon_4$
Pnondet
$\varepsilon_5 \underset{C_5^{conv}}{\leftrightarrow} x$
$\varepsilon_6 \underset{C_6^{conv}}{\leftrightarrow} y$

Figure 7: The input/output diagram of Pnondet

The informal description of the intended semantics of the program Pnondet is as follows: each time a set of values $\{\underline{m_1},\underline{n_1},\underline{m_2},\underline{n_2}\}$ is fed into Pnondet,

1. it randomly chooses suitable values $\underline{m_3}$ and $\underline{n_3}$ for m_3 and n_3, respectively, satisfying the conditions imposed on them by $CONV^{EEQ^+}$;
2. then, it chooses, also randomly, any pair $\langle \underline{x},\underline{y} \rangle$ of coordinates for a point lying inside the convex polygon defined by the constraints $y \leq \underline{m_1}x + \underline{n_1}$, $y \geq \underline{m_2}x + \underline{n_2}$, $y \leq \underline{m_3}x + \underline{n_3}$, $x > 0$, and $y > 0$ also imposed by $CONV^{EEQ^+}$;
3. finally, it outputs the values $\underline{x}$ and $\underline{y}$.

Thus, Pnondet has an obviously non-deterministic behaviour, since, after randomly choosing values for the coefficients of the straight line $y = m_3x + n_3$, (thus defining a particular convex polygon), it produces as output an also randomly chosen pair $\langle \underline{x},\underline{y} \rangle$ of coordinates defining a point lying inside this convex polygon.

Now, let us suppose we fed Pnondet 21 consecutive times with values $\underline{m_1} = 0.5$, $\underline{n_1} = 4$, $\underline{m_2} = 2$, and $\underline{n_2} = -8$. As we said above, each time (i.e., in each execution), Pnondet will first randomly choose values $\underline{m_3}$ and $\underline{n_3}$, thus defining a particular convex polygon.

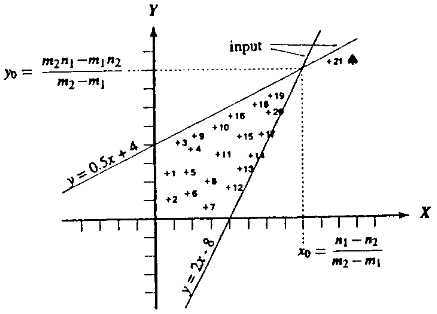


Figure 8: Output generated by 21 executions of Pnondet

In each one of these 21 executions, Pnondet produces an output pair of values $\langle \underline{x},\underline{y} \rangle$ defining the points depicted in Fig. **8** (see also in the table in Fig. **9**, columns m_1, n_1, m_2, n_2, x, and y, which exhibit the input/output relation defined by the 21 executions of Pnondet in question).

The purpose of this example is to test (using the variant of Schema I for non-deterministic programs) whether or not the behaviour of `Pnondet` satisfies its intended semantics (and, therefore, if it is correct with respect to $CONV^{EEQ^+}$). Thus, what we should do now is to test each one of the axioms of $CONV^{EEQ^+}$ (on the basis of $CONV^*$), using the input/output relation produced by the 21 executions of `Pnondet` as evidence. To do this, we need to derive, for each one of the essential variables z of each one of these axioms, an appropriate $CONV^*_z$. However, it will suffice as an example to analyse only the testing of $H_1 : m_3 < 0$ and $H_2 : n_3 \leq k$.

input					output		$H_1 : m_3 < 0$	$H_2 : n_3 \leq k$ (a)		
m_1	n_1	m_2	n_2	point number	[illegible]	[illegible]	$CONV_{m_3} : m_3 < m_1$	n_3 lowerbound calculated with: $CONV_{n_3}$: [illegible] $< n_3$	m_3 upperbound calculated with: $CONV_{m_3}$: $m_3 < 0$	k least upperbound calculated using $m_3 = 0$
0.5	4	2	-8	1	[illegible]	[illegible]	0.5	2.45	0	8
0.5	4	2	-8	2	[illegible]	[illegible]	0.5	1.05	0	8
0.5	4	2	-8	3	[illegible]	[illegible]	0.5	4.10	0	8
0.5	4	2	-8	4	[illegible]	[illegible]	0.5	3.75	0	8
0.5	4	2	-8	5	[illegible]	[illegible]	0.5	2.50	0	8
0.5	4	2	-8	6	[illegible]	[illegible]	0.5	1.35	0	8
0.5	4	2	-8	7	[illegible]	[illegible]	0.5	0.60	0	8
0.5	4	2	-8	8	[illegible]	[illegible]	0.5	2.00	0	8
0.5	4	2	-8	9	[illegible]	[illegible]	0.5	4.45	0	8
0.5	4	2	-8	10	[illegible]	[illegible]	0.5	4.90	0	8
0.5	4	2	-8	11	[illegible]	[illegible]	0.5	3.45	0	8
0.5	4	2	-8	12	[illegible]	[illegible]	0.5	1.65	0	8
0.5	4	2	-8	13	[illegible]	[illegible]	0.5	2.65	0	8
0.5	4	2	-8	14	[illegible]	[illegible]	0.5	3.35	0	8
0.5	4	2	-8	15	[illegible]	[illegible]	0.5	4.40	0	8
0.5	4	2	-8	16	[illegible]	[illegible]	0.5	5.50	0	8
0.5	4	2	-8	17	[illegible]	[illegible]	0.5	4.50	0	8
0.5	4	2	-8	18	[illegible]	[illegible]	0.5	6.10	0	8
0.5	4	2	-8	19	[illegible]	[illegible]	0.5	6.55	0	8
0.5	4	2	-8	20	[illegible]	[illegible]	0.5	5.65	0	8
♠ 0.5	4	2	-8	21	[illegible]	[illegible]	0.5	8.35	0	8

(a) with $k = \dfrac{m_2 n_1 - m_1 n_2 + m_3(n_2 - n_1)}{m_2 - m_1}$

Figure 9: Data for the testing of hypotheses H_1 and H_2

So, let us begin with $H_1 : m_3 < 0$. It is obvious that it has just one essential variable, i.e., m_3. Therefore, we need to derive only a subtheory $CONV^*_{m_3}$. Notice that for this derivation, we can use all the axioms of $CONV^{EEQ^+}$ with the exception of $m_3 < 0$ itself, because we would otherwise violate condition (iii) of Schema I. To see why, suppose we allow the use of $m_3 < 0$ in the derivation of $CONV^*_{m_3}$, then it is obvious that $\&CONV \vdash H_1$. This leads to the violation of the said condition (iii) because for any ground tautology K in $CONV^*$, since K has no variables and hence $Evar(K) \subset Evar(H_1) = \{m_3\}$, obviously $\&CONV, K \vdash H_1$ and $\&CONV, H_1 \vdash K$ holds.

Then, let us use the inequalities

$$n_1 < n_3 \leq \frac{m_2 n_1 - m_1 n_2 + m_3(n_2 - n_1)}{m_2 - m_1}$$

From them we obtain that

$$\begin{aligned}
n_1 &< \tfrac{m_2n_1 - m_1n_2 + m_3(n_2-n_1)}{m_2-m_1} \\
(m_2-m_1)n_1 &< m_2n_1 - m_1n_2 + m_3(n_2-n_1) \text{ since } m_2 - m_1 > 0 \\
m_2n_1 - m_1n_1 &< m_2n_1 - m_1n_2 + m_3(n_2-n_1) \\
m_1n_2 - m_1n_1 &< m_3(n_2-n_1) \\
m_1(n_2-n_1) &< m_3(n_2-n_1) \\
m_1 &> m_3 \text{ since } n_2 - n_1 < 0
\end{aligned}$$

that is,

$$CONV_{m_3} : m_3 < m_1$$

which is a presentation of $CONV^*_{m_3}$.

In the table of Fig. 9 we have one column under the key $H_1 : m_3 < 0$. Notice that $CONV_{m_3}$ is of the type referred to in case 3 (on page 14). Therefore $\underline{m_3} = (-\infty, 0.5]$ and the $H_1 : m_3 < 0$ is satisfied since there exists a value in $\underline{m_3}$ smaller than 0.

Let us now test the hypothesis $H_2 : n_3 \leq k = \frac{m_2n_1 - m_1n_2 + m_3(n_2-n_1)}{m_2-m_1}$ on the basis of $CONV^*$. The set $Evar(H_2)$ of essential variables of H_2 is $\{m_1, n_1, m_2, n_2, m_3, n_3\}$. Given that m_1, n_1, m_2, n_2 are part of the evidence, their corresponding subtheories can functionally determine them. So, we set

$CONV_{m_1} : m_1 = m_1$

$CONV_{n_1} : n_1 = n_1$

$CONV_{m_2} : m_2 = m_2$, and

$CONV_{n_2} : n_2 = n_2$

From $y \leq m_3x + n_3$, and given that, according to $CONV$, n_3 as well as x are positive and m_3 negative, we can deduce that $y < n_3$. The subtheories for m_3 and n_3 are then:

$CONV_{m_3} : m_3 < 0$, and

$CONV_{n_3} : y < n_3$

Recall that a subtheory $\mathcal{T}^*_z$ must constrain the value of z using only the evidence. Therefore, in the case of m_3, we cannot use the inequality $y \leq m_3x + n_3$, since we cannot eliminate n_3, even using other parts of $CONV$.

In the table of Fig. 9 we have three columns, now under the key $H_2 : n_3 \leq k$ giving bounds for the values of n_3, of m_3, and of k.

Notice that $CONV_{n_3}$ is of the type referred to in case 2 (on page 14). Therefore $\underline{n_3} = (8.35, \infty)$ and $H_2 : n_3 \leq k$ is not satisfied since there exists a value in $\underline{m_3}$ that makes k smaller than 8.35. (See column 'k lower bound using $m_3 = 0$,' calculated using the least upper bound 0 for m_3.) This is due to the point number 21 (labelled with ♠ both in Fig. **8** and in the table in Fig. **9**), which is obviously outside any possible convex polygon.

Therefore, `Pnondet` is not correct with respect to $CONV^*$.

If we analyse the table of Fig. **9**, we can consider the upperbound calculated for m_3 in the test of $H_1 : m_3 < 0$ to be too coarse, because for any point i in Fig. **8**, the least upperbound for m_3 –not even considering in its calculation the hypothesis H_1– will be 0. Since the point i will obviously be inside the convex polygon limited by L_1, L_2, the coordinate axes, and the line given by $y = y_i$, $m_3 = 0$ will be a finer upperbound. However, we are testing `Pnondet` against the whole theory. Thus, we should also consider that for this point i, we have a greatest lowerbound for n_3 namely $n_3 = y_i$. Therefore, the line limiting the minimum polygon in which the point i lies will be $L : y = mx + n$ with $m < m_1$ and $n > y_i$; this line L is $y = y_i$.

7 Conclusions

What we have presented is a very promising and powerful approach to specification-based verification and validation testing of software process artefacts (i.e., software artefacts as well as specifications). We have shown that the approach is useful for both deterministic and non-deterministic programs and specifications.

This approach was developed using the general epistemological background sketched in Sects. 1, 3, and 5. We would like to emphasise that in doing this, as theoretical computer scientists, we proceeded *more like epistemologists than as logicians or mathematicians* in conducting our quest.

On the same basis, in Sect. 2 we have also discussed the flaws of the best known alternative to the bootstrap strategy, namely the hypothetico-deductive strategy. These flaws prevent its use for sustaining testing methodologies.

Whether in the framework of verification testing, it is worth noting the similarity between the bootstrap core idea and the notion of implementation relation defined by refinement. Recall in Fig. **1** the valuation α induced by $\underline{E}$, that has to be such that $\alpha \models H$; in other words, the bootstrap strategy requires that evidence provide instances of the hypothesis under test. Refinement requires that the relation defined by (the semantics of) a program be contained in the set of relations defined by (the semantics of) a specification; see [1]. Thanks to the deduction theorem, both just mean that evidence resp. program implies the specification.

As we said in the introduction, 'exhaustive' testing has two different dimensions. One is the 'coverage' by the test of the whole specification to be tested. The other is the 'exhaustion' of the test w.r.t. the (possibly infinite) domain of interpretation of the symbols in the specification. It seems that our approach takes care of the first of these dimensions. However, we have presented it just for theories over an existential (in)equational logic. Then, the bootstrap-strategy-based approach should be extended to deal with theories over other logics of interest (e.g., classical first-order, temporal, deontic, and dynamic logics) and with various kinds of semantics (for instance, transition systems). This is a must if we intend to turn the approach to a practical one. Also the bootstrap testing of structured specifications and systems must be considered. The second dimension (which is the problem addressed by Gaudel with her 'uniformity hypotheses') must be studied, for instance, in the light of Carnap's and Hintikka's results on inductive logic and the general methodology of (scientific) induction [4, 13].

Comparing the bootstrap strategy with real software testing developments is a crucial issue. This comparison should begin by an exhaustive analysis of the current testing methods in the light of the epistemological framework here presented. A starting point could be, for instance, the method reported in [6], whose similarity with this strategy was pointed out by C. Heitmeyer.

References

[1] M. Bidoit, M. V. Cengarle, and R. Hennicker. Proof systems for structured specifications and their refinements. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*. Springer-Verlag, 1999.

[2] R. Carnap. *Der logische Aufbau der Welt*. Weltkreis-Verlag, Berlin, 1928.

[3] R. Carnap. *Philosophical foundations of physics*. Basic books, New York, 1966.

[4] R. Carnap. *Logical Foundations of Probability*. Univ. of Chicago Press, Chicago–Illinois, 1977. 2nd rev. edition, 1962.

[5] M. V. Cengarle and A. M. Haeberer. Towards an epistemology-based methodology for verification and validation testing. Technical Report 0001, LMU München, Inst. für Informatik, Jan. 2000. 71 pages.

[6] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In O. Nierstrasz and M. Lemoine, editors, *Proc. of ESEC/FSE'99*, volume 1687 of *LNCS*. Springer-Verlag, May 1999.

[7] M.-C. Gaudel. Testing can be formal, too. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *Proc. of TAPSOFT'95*, volume 915 of *LNCS*, pages 82–96. Springer-Verlag, May 1995.

[8] M.-C. Gaudel and P. R. James. Testing data types and processes, an unifying theory. In *3rd ERCIM on FMICS'98*. CWI, May 1998.

[9] M.-C. Gaudel and P. R. James. Testing algebraic data types and processes: a unifying theory. *Formal Aspects of Computing*, 1999. To appear.

[10] C. Glymour. Hypothetico-deductivism is hopeless. *Philosophy of science*, 47:322–325, 1980.

[11] C. Glymour. *Theory and Evidence*. Princeton Univ. Press, New Jersey, 1980.

[12] A. M. Haeberer and T. S. E. Maibaum. The very idea of software development environments: A conceptual architecture for the arts environment. In B. Nuseibeh and D. Redmiles, editors, *Proc. of ASE'98*, pages 260–269. IEEE CS Press, 1998.

[13] J. Hintikka. Toward a theory of inductive generalization. In Y. Bar-Hillel, editor, *Proceedings of the 1964 Congress for Logic, Methodology, and the Philosophy of Science*, pages 274–288, Amsterdam, 1962. Stanford Univ. Press.

[14] M. M. Lehman. *Program Evolution – Processes of Software Change*. Academic Press, New York, 1985. ISBN 0-12-442441-4.

[15] G. H. Merrill. Confirmation and prediction. *Philosophy of science*, 46:98–117, 1979.

[16] C. K. Waters. Relevance logic brings hope to hypothetico-deductivism. *Philosophy of science*, 54:453–464, 1987.

Using CASL to Specify the Requirements and the Design: A Problem Specific Approach

Christine Choppy[1] and Gianna Reggio[2]

[1] LIPN, Institut Galilée - Université Paris XIII, France
[2] DISI, Università di Genova, Italy

Abstract. In [11] M. Jackson introduces the concept of *problem frame* to describe specific classes of problems, to help in the specification and design of systems, and also to provide a framework for reusability. He thus identifies some particular frames, such as the translation frame (e.g., a compiler), the information system frame, the control frame (or reactive system frame), ... Each frame is described along three viewpoints that are application domains, requirements, and design.
Our aim is to use CASL (or possibly a sublanguage or an extension of CASL if and when appropriate) to formally specify the requirements and the design of particular classes of problems ("problem frames"). This goal is related to methodology issues for CASL, that are here addressed in a more specific way, having in mind some particular problem frame, i.e., a class of systems.
It is hoped that this will provide both a help in using, in a really effective way, CASL for system specifications, a link with approaches that are currently used in the industry, and a framework for the reusability.
This approach is illustrated with some case studies, e.g., the information system frame is illustrated with the invoice system.

1 Introduction

It is now well established that formal specifications are required for the development of high quality computer systems. However, it is still difficult for a number of practitioners to write these specifications. In this paper we address the general issue of how to bridge the gap between a problem requirements and its specification. We think this issue has various facets. For instance, given a problem, how to guide the specification process? Often people do not know where to start, and then are stopped at various points. Another facet is, given a specification language, how to use it in an appropriate way? We address these facets here through the use of M. Jackson's problem frames (successfully used in industry), which we formalize by providing the corresponding specification skeletons in the CASL language ([12, 13]).

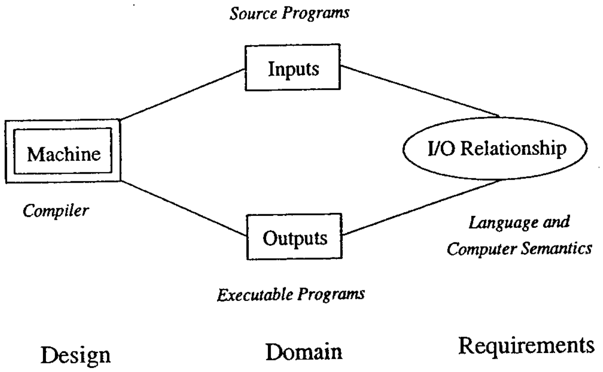
A Jackson problem frame [11] is a generalization of a class of problems, thus helping to sort out in which frame/category is the problem under study. The idea here is to provide a help to analyse software development problems and to choose an appropriate method for solving them (with the assumption that there is no "general" method). Then, for each problem frame, M. Jackson provides its expected components together with their characteristics and the way they are connected. The problem frame components always include a domain description, the requirements description, and possibly the design description. The domain description expresses "what already exists", it is an abstraction of a part of the world. The requirements description expresses "what there should be", that is what are the computer system expected concepts and facilities. The design description deals with "how" to achieve the system functions and behaviours. The problem frames identified are the translation (JSP), the information system, the reactive system (or control frame), the workpiece, and the connection frames. While these cover quite a number of applications, it may also be the case that some problems are "multiframe".

While Jackson problem frames may be used to start understanding and analysing the system to be developed, they have no formal underpinning. Our idea here is to still rely on them while providing the corresponding "specification frames". We thus provide a methodological approach to write problem specifications using the problem frames that importantly gives guidelines to *start* and to *do* the problem analysis and specification; combining these two "tools" yields a powerful approach to guide the problem understanding and the specification writing.

The issue of the choice of a formal specification language is non trivial [2]. We think that algebraic specification languages offer an adequate degree of abstraction for our needs, so we chose the latest and more general one, CASL, the Common Algebraic Specification Language ([12, 13]) developed within the Common Framework Initiative (CoFI), to express our proposed formal underpinning for problem frames. While CASL was developed to be "a central, reasonably expressive language for specifying conventional software", "the common framework will provide a family of languages" [12]. Thus, restrictions of CASL to simpler languages (for example, in connection with verification tools) will be provided as well as extensions oriented towards particular programming paradigms, e.g., reactive systems. While dealing with the translation frame (Sect. 2 and 3), CASL complies with our needs, but when moving to dynamic systems that may occur within the information system frame (Sect. 4 and 5), we propose an extension of CASL with temporal logic, named CASL-LTL [15], based on the ideas of [3] and [7], that may be more appropriate (and will be shortly presented when used). Since the design of CASL was based on a critical selection of constructs found in existing algebraic frameworks, the reader familiar with these may feel quite "at home", while CASL offers for these convenient syntactic combinations. In this paper, we use some CASL constructs, that we introduce when they appear. While the CASL syntax and semantics are completed, some tools (e.g., parsers, libraries, ...) are being developed. In what follows we shall rely on the available library for basic data types [16].

This paper is organised as follows. In Sect. 2 and 3, we describe the translation problem frame, provide a a method to formalize it using CASL, and illustrate it on a short example that is the Unix grep utility. In Sect. 4 and 5, we work similarly on the information system problem frame and illustrate it with the invoice case study. The information system frame raises various issues, since we are dealing with bigger reactive systems. The size issue leads us to clearly identify sets of properties that need to be expressed and also to search for a legible way to present large specifications of this kind. For lack of room, we cannot report here the complete specifications of the considered case studies, they can be found in [6].

2 Translation Frame



The translation frame we consider here is a simple frame that is quite useful for a number of case studies [5] (it is close to the JSP frame where inputs and outputs are streams). The translation frame domain is given by the Inputs and the Outputs, the requirements are described by the input/output relationship, I/O Relationship, and the design is the Machine. An example of a translation frame problem is a compiler, where the Inputs are the source programs, the Outputs are the executable programs, the I/O Relationship is given by the language and computer semantics, and the Machine is the compiler. In the following, we shall provide the skeletons for the CASL formal specifications of Inputs, Outputs, the I/O Relationship, and the Machine, as well as conditions for correctness of the Machine as regards the I/O Relationship. This will be shortly illustrated on a case study (the Grep utility) in Sect. 3.

2.1 Domain and Requirements

To capture the requirements in this case means:
- to express the relevant properties of the application domain components, i.e., Inputs and Outputs;
- to express the I/O Relationship.
Let us note that this often yields to specify also some basic data that are required by the Inputs, the Outputs and/or by the I/O Relationship.

To use the CASL language to specify the above requirements means to give three CASL specifications of the following form:

```
spec INPUTS =
    ......

spec OUTPUTS =
    ......
```

```
spec IO_RELATIONSHIP =
    INPUTS and OUTPUTS then
    pred    IO_rel : Input × Output
    axioms
    ......
```

where the IO_RELATIONSHIP specification extends (CASL keyword **then**) the union (**and**) of the INPUTS and OUTPUTS specifications by the *IO_rel* predicate. The axioms in CASL are first-order formulas built from equations and definedness assertions. We can here add some suggestions on the way the axioms of IO_RELATIONSHIP should be written. The *IO_rel* properties could be described along the different cases to be considered and expressed by some conditions on the *Input* and *Output* arguments. This approach has the advantage that the specifier is induced to consider all relevant cases (and not to forget some important ones). Therefore the axioms of IO_RELATIONSHIP have the form

either $IO\_rel(i,o) \Rightarrow cond(i,o)$

or $cond(i,o) \wedge def\ i \wedge def\ o \Rightarrow IO\_rel(i,o)$.

where i and o are terms of appropriate sorts and *cond* is a CASL formula.

2.2 Design

To design a solution of the problem in this case means to define a partial function (or a sequential program or an algorithm) *transl* that associates an element of Outputs with an element of Inputs. To use CASL to specify the above design means to give a CASL specification of the following form:

```
spec MACHINE =
    INPUTS and OUTPUTS then
free {  %% this CASL "free" construction requires that no additional feature occurs
    op  transl : Input →? Output   %% the translation function
    axioms
    %% transl domain definition
    ......
    %% transl definition
    ......}
```

The axioms for *transl* should exhibit both (i) when is *transl* defined (*transl* domain definition), and (ii) what are *transl* results (*transl* definition).

Again here, we suggest a case analysis approach which yields for the *transl* domain definition axioms of the form

$cond(i) \Rightarrow def\ transl(i)$

and for the *transl* definition axioms of the form

$cond(i,o) \wedge def\ (transl(i)) \wedge def\ o \Rightarrow transl(i) = o$

where i and o are terms of the appropriate sorts, and *cond* is a positive conditional formula. Let us note that, in order to provide a more concise/readable presentation of the axioms, the $def\ (transl(i)) \wedge def\ o$ part may be left implicit.

2.3 Correctness

Here we add some notion of correctness which is not explicited in Jackson's presentation, and which we can deal with thanks to the formalization we provide. It may now be relevant to state under which conditions the Machine designed implements the IO_Relationship. We propose below three conditions and introduce the following specification, requiring that the predicate *IO_rel* does not belong to the Machine signature.

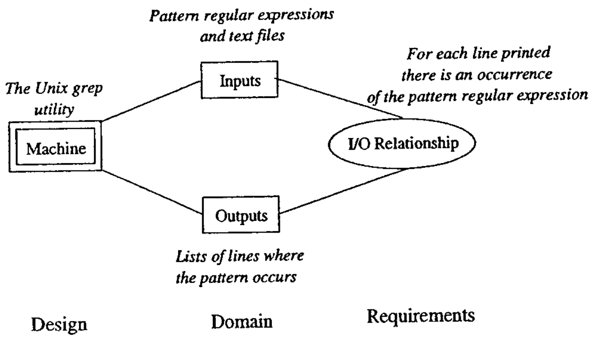
spec Translation = IO_Relationship **and** Machine

Machine is correct w.r.t. IO_Relationship iff

1. Machine is sufficiently complete and hierarchically consistent w.r.t. Inputs and Outputs.
2. Translation $\models \forall\ i : Input, o : Output, transl(i) = o\ \Rightarrow\ IO\_rel(i, o)$
3. Translation $\models \forall\ i : Input, o : Output, IO\_rel(i, o)\ \Rightarrow$ $\exists\ o' : Output \bullet transl(i) = o'$

Condition 1 requires that Machine does not introduce some new elements or properties in the specified descriptions of "what already exists" (the application domain), i.e., Inputs and Outputs. Condition 2 requires that, whenever *transl* is defined for a given i and yields o, then IO_Relationship relates i and o, in other words, it ensures that the produced translation is correct. Finally, condition 3 expresses that whenever IO_Relationship relates i with some o (recall *IO_rel* is just a relationship not a function), then *transl* applied to i must yield some o', in other words, it requires that the translation produces an output when appropriate given the requirements.

3 Case Study: The Grep Operation



In the previous section, the translation frame was presented with the typical compiler example. Here, we illustrate it with the grep utility that is provided by Unix and we sketch the corresponding specifications (see [6] for the full ones). The grep utility searches files for a pattern and prints all lines that contain that pattern. It uses limited regular expressions to match the patterns.

3.1 Domain and Requirements

In order to provide a specification of the domain, we need to specify the inputs, which are regular expressions and files, the outputs, which are lists of lines, and also the basic data that are required, which are characters, strings and lists.

Basic Data To specify the basic data we use some specifications provided in [16], e.g., CHAR and STRING. For example the specification for strings of [16] is an instantiation of the generic specification LIST[ELEM] together with some symbol mapping (↦) for sort names:

```
spec STRING = LIST[CHAR]
   with sorts List[Char] ↦ String
```

Inputs We sketch below the specifications of the Inputs which are regular expressions and files.

```
spec GREP_INPUTS = REGULAR_EXPRESSION and FILE
```

The CASL construct **free type** allows one to provide for the *Reg_Expr* type constants (*empty* and *Λ*), operations (__ + __ and __*), and also to state that any character may yield a *Reg_Expr*.

```
spec REGULAR_EXPRESSION = CHAR then
free  type Reg_Expr ::=
  empty | Λ | __ + __ : (Reg_Expr Reg_Expr) | __* : (Reg_Expr) | sort Char ;

spec FILE = STRING then
free  type File ::= empty | __ __ : (Char ; File);
  ops first_line : File →? String;
      drop_line : File →? File;
%% with the corresponding axioms
```

Outputs is a list of lines, that is a list of strings.

```
spec GREP_OUTPUTS = LIST[STRING]
   with sorts List[String] ↦ Grep_Output
```

I/O Relationship The I/O Relationship between the Inputs and the Outputs is sketched in the following specification, where the *grep_IO_rel* predicate properties are expressed by means of the predicates *is_gen* (stating when a string matches a regular expression) and *appears_in* (stating when a string is a substring of another one).

```
spec GREP_IO_REL = GREP_INPUTS and GREP_OUTPUTS
then preds grep_IO_rel : Reg_Expr × File × Grep_Output;
      __ is_gen __ : String × Reg_Expr;
      __ appears_in __ : String × String;
  vars reg : Reg_Expr; ol, ol' : Grep_Output; f : File;
```

axioms
$grep\_IO\_rel(reg, empty, ol) \Leftrightarrow ol = nil;$
$\neg f = empty \Rightarrow$
$(grep\_IO\_rel(reg, f, ol) \Leftrightarrow$
$((\exists s \bullet is\_gen\ (reg, s) \wedge s\ appears\_in\ first\_line(f) \wedge$
$grep\_IO\_rel(reg, drop\_line(f), ol') \wedge ol = first\_line(f) :: ol')$
$\vee (\neg \exists s \bullet is\_gen\ (reg, s) \wedge s\ appears\_in\ first\_line(f) \wedge$
$grep\_IO\_rel(reg, drop\_line(f), ol)));$
%% axioms defining *appears_in* and *is_gen*

3.2 Design

The MACHINE yields an *Grep_Output* given a *Reg_Expr* and a *File*.

spec GREP_MACHINE = GREP_INPUTS **and** GREP_OUTPUTS
then op $grep\_transl : Reg\_Expr \times File \rightarrow Grep\_Output;$
pred $match : Reg\_Expr \times String;$
vars $reg : Reg\_Expr;\ f : File;$
axioms
$grep\_transl(reg, empty) = empty;$
$\neg (f = empty) \wedge match(reg, first\_line(f)) \Rightarrow$
$grep\_transl(reg, f) = first\_line(f) :: grep\_transl(reg, drop\_line(f));$
$\neg (f = empty) \wedge \neg match(reg, first\_line(f)) \Rightarrow$
$grep\_transl(reg, f) = grep\_transl(reg, drop\_line(f));$
%% axioms defining *match*

3.3 Correctness

To express correctness we need to introduce the following specification, requiring that the predicate *grep_IO_rel* does not belong to the GREP_MACHINE signature.

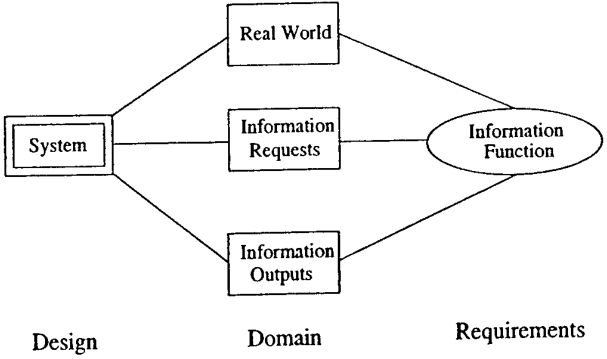
spec GREP_TRANSLATION = GREP_IO_REL **and** GREP_MACHINE

GREP_MACHINE is correct w.r.t. GREP_IO_REL iff

1. GREP_MACHINE is sufficiently complete and hierarchically consistent w.r.t. GREP_INPUTS and GREP_OUTPUTS.
2. GREP_TRANSLATION $\models \forall f : File, reg : Reg\_Expr, ol : Grep\_Output,$ $grep\_transl(reg, f) = ol \Rightarrow grep\_IO\_rel(reg, f, ol)$
3. GREP_TRANSLATION $\models \forall f : File, reg : Reg\_Expr, ol : Grep\_Output,$ $grep\_IO\_rel(reg, f, ol) \Rightarrow \exists ol' : Grep\_Output \bullet grep\_transl(reg, f) = ol'$

4 Information System Frame

The information system frame domain description is given by the Real World, the Information Requests and the Information Outputs, the requirements are described by the Information Function, and the design is the System. To quote [11], "In its simplest form, an information system provides information, in response to requests, about some relevant real-world domain of interest." The Real World may be a *static* domain (e.g., if the system provides information on Shakespeare's plays), or a *dynamic* domain (e.g., "the activities of a currently operating business" [11]). Here we consider information system frames with a dynamic domain, so "The Real World is *dynamic* and also *active.*" [11].



4.1 Domain and Requirements

To capture the requirements in the case of the Simple Information System means:

- to find out the relevant properties of the Real World;
- to determine the Information Requests and the Information Outputs;
- to determine the Information Function.

To use Casl-Ltl [15] to specify the above requirements means to give four specifications corresponding to the four parts respectively, as follows.

We consider the case where the Real World is a dynamic system, thus is specified using Casl-Ltl by logically specifying an lts (a *labelled transition system*) that models it. A *labelled transition system* (shortly *lts*) is a triple $(S, L, \rightarrow)$, where S and L are two sets, and $\rightarrow\subseteq S \times L \times S$ is the *transition relation*. A triple $(s, l, s') \in\rightarrow$ is said to be a *transition* and is usually written $s \xrightarrow{l} s'$. Using the **dsort** construction introduced in Casl-Ltl is a way to declare the lts triple $(S, L, \rightarrow)$ at once and to provide the use of temporal logic combinators in the axioms (as defined in Casl-Ltl).

Given an lts we can associate with each $s_0 \in S$ the tree (*transition tree*) whose root is s_0, where the order of the branches is not considered, two identically decorated subtrees with the same root are considered as a unique subtree, and if it has a node n decorated with s and $s \xrightarrow{l} s'$, then it has a node n' decorated with s' and an arc decorated with l from n to n'.

We model a dynamic system D with a transition tree determined by an lts $(S, L, \rightarrow)$ and an initial state $s_0 \in S$; the nodes in the tree represent the intermediate (interesting) situations of the life of S, and the arcs of the tree the possibilities of S of passing from one situation to another. It is important to note here that an arc (a transition) $s \xrightarrow{l} s'$ has the following meaning: D in the situation s has the *capability* of passing into the situation s' by performing a transition, where the label l represents the interaction with the environment during such a move; thus l contains information on the conditions on the environment for the capability to become effective, and on the transformation of such environment induced by the execution of the transition.

We assume that the labels of the lts modelling the Real World are finite sets of events, where an *event* is a fact/condition/something happening during the system life that is relevant to the considered problem. So we start by determining which are the events and by classifying them with a finite number of kinds, then we specify them with a simple CASL specification of a datatype, where any kind of event is expressed by a generator.

```
spec EVENT =
    ... then
    free type  Event ::=  ...
```

At this stage it is not advisable to precisely specify the states of the lts modelling the Real World; however, we need to know something about them, and we can express that by some CASL operations, called *state observers*, taking the state as an argument and returning the observed value. Finally we express the properties on the behaviour of the Real World along the following schema:

- **Incompatible events:** Express when sets of events are incompatible, i.e., when they cannot happen simultaneously.
- **Relationships between state observers and events** For each state observer *obs* express (i) its initial value, (ii) its value after $E(\ldots)$ happened, for each kind of event E modifying it.
- **Event specific properties** For each kind of event E express the
    - **preconditions** properties on the system state and on the past (behaviour of the system) required for $E(\ldots)$ to take place
    - **postconditions** properties on the state and on the future (behaviour of the system) that should be fulfilled after $E(\ldots)$ took place
    - **liveness** properties on the state under which $E(\ldots)$ will surely happen and when it will happen.

Then the specification of the Real World has the following form (the specification FINITESET has been taken from [16]).

```
spec REAL_WORLD =
    FINITESET[EVENT fit Elem ↦ Event] ... then
    dsort  State %% dsort is a CASL-LTL construction
    free type  Label_State ::= sort FinSet[Event];
    pred     initial : State   %% determines the initial states of the system
    %% State observers
    op  obs : State× ... → ...
    ......
    axioms
    %% Incompatible events
    ...
    %% Relationships between state observers and events
```

```
...
%% Event specific properties
...
```

The postconditions and the liveness properties cannot be expressed using only the first-order logic available for axioms in CASL, thus CASL-LTL extends it with combinators from the temporal logic ([7]) that will be introduced when they will be used in the case study.

The Information Requests and the Information Outputs are two datatypes that are specified using CASL by simply giving their generators.

```
spec INFORMATION_REQUESTS =          spec INFORMATION_OUTPUTS =
    ... then                              ... then
    free type Info_Request ::= ...        free type Info_Output ::= ...
```

We assume that the Information Function takes as arguments, not only the information request, but also the history of the system (a sequence of states and labels), because it contains all the pieces of information needed to give an answer.

The Information Function is specified using CASL-LTL by defining it within a specification of the following form.

```
spec INFORMATION_FUNCTION =
    INFORMATION_REQUESTS and INFORMATION_OUTPUTS and REAL_WORLD then
free {
    %% histories are partial system lifecycles
    type History ::= init(State) | ___(History; Label_State; State)?;
    op  last : History → State;
    vars  st : State; h : History; l : Label_State;
    •   def (h l st)  ⟺  last(h) --l--> st   %% ___ is partial
    •   last(init(st)) = st
    •   def (h l st)  ⇒  last(h l st) = st
    op  inf_fun : History×Info_Request → Info_Output;
    axioms
    %% properties of inf_fun ...
```

where the properties of *inf_fun* are expressed by axioms having the form
$def\ (h) \wedge def\ (i\_req) \wedge def\ (i\_out) \wedge cond(h, i\_req, i\_out) \Rightarrow$
$inf\_fun(h, i\_req) = i\_out$
and *cond* is a conjunction of positive atoms.

In many cases the above four specifications share some common parts, by using the CASL constructs for the declaration of named specifications, such parts can be specified apart and reused when needed. These specifications are collected together and presented before the others under the title of basic data.

4.2 Design

To design an "Information System" means to design the System, a dynamic system interacting with the Real World (by detecting the happening events), and with the users (by receiving the information requests and sending back the information outputs).

We assume that the System:

- keeps a view of the actual situation of the Real World,
- updates it depending on the detected events,
- decides which information requests from the users to accept in each instant,
- answers to such requests with the appropriate information outputs using its view of the situation of the Real World.

We assume also that the System can immediately detect in a correct way any event happening in the Real World and that the information requests are handled immediately (more precisely the time needed to detect the events and to handle the requests is not relevant).

The design of the System will be specified using CASL-LTL by logically specifying an lts that models it. The labels of this lts are triples consisting of the events detected in the Real World, the received requests and the sent out information output.

```
spec SYSTEM =
    SITUATION and FINITESET[EVENT fit $Elem \mapsto Event$] and
    FINITESET[INFORMATION_REQUESTS fit $Elem \mapsto Info\_Request$] and
    FINITESET[INFORMATION_OUTPUTS fit $Elem \mapsto Info\_Output$] then
free {
    dtype
        $System$ ::= sort $Situation$;
        $Label\_System$ ::= $\_\_\_(FinSet[Event];$
            $FinSet[Info\_Request]; FinSet[Info\_Output]);$
    ops $update : Situation \times FinSet[Event] \to Situation;$
        $inf\_fun : Situation \times Info\_Request \to Info\_Output;$
    pred    $acceptable : FinSet[Info\_Request];$
    axioms
        $i\_reqs = \{i\_req_1\} \cup \ldots \cup \{i\_req_n\} \ \wedge \ acceptable(i\_reqs) \wedge$
        $i\_outs = \{inf\_fun(sit, i\_req_1)\} \cup \ldots \cup \{inf\_fun(sit, i\_req_n)\} \ \Rightarrow$
            $sit \xrightarrow{evs\ i\_reqs\ i\_outs} update(sit, evs);$
    %% axioms defining update, acceptable and inf_fun
        ......
}
```

where SITUATION specifies a data structure describing in an appropriate way (i.e., apt to permit to answer to all information requests) the System's views of the possible situations of the Real World.

Thus to specify the design of the System it is sufficient to give:

- the specification SITUATION;
- the axioms defining the operation *update* describing how the System updates its view of the Real World when it detects some events;
- the axioms defining the predicate *acceptable* describing which sets of requests may be accepted simultaneously by the System;
- the axioms defining the operation *inf_fun* describing what is the result of each information request depending on the System's view of the the Real World situation.

4.3 Correctness

We introduce the following specification:

spec INFORMATION_SYSTEM =
  INFORMATION_FUNCTION **and** SYSTEM **then**
  pred $Imp : History \times Situation$
  axioms
  ......

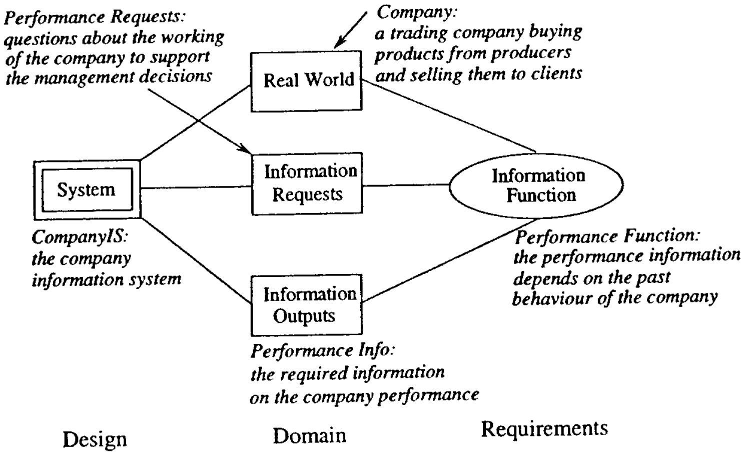
SYSTEM is correct w.r.t. INFORMATION_FUNCTION iff

1. SYSTEM is sufficiently complete and hierarchically consistent w.r.t. EVENT, INFORMATION_REQUESTS and INFORMATION_OUTPUTS.
2. INFORMATION_SYSTEM $\models \forall\ st : Situation, i\_req : Info\_Request,$
   $i\_out : Info\_Output, inf\_fun(st, i\_req) = i\_out \Rightarrow$
   $\exists\ h : History \bullet Imp(h, st) \wedge inf\_fun(h, i\_req) = i\_out$

Notice that the proof has to be done in the realm of the first-order logic, and not require to consider the temporal extenxion of CASL-LTL, in this frame the temporal combinators are used only to express the properties of the Real World, i.e., of the application domain.

5 Case Study: The Invoice System

5.1 The Invoice System


Performance Requests: questions about the working of the company to support the management decisions
Company: a trading company buying products from producers and selling them to clients
Real World
System
Information Requests
Information Function
CompanyIS: the company information system
Performance Function: the performance information depends on the past behaviour of the company
Information Outputs
Performance Info: the required information on the company performance
Design
Domain
Requirements

This case study, the invoice system, is inspired from one proposed in [1]. The problem under study is an information system for a company selling products to clients to support the management decisions. The clients send orders to the company, where an order contains one and only one reference to an ordered product in a given quantity. The status of an order will be changed from "pending" to "invoiced" if the ordered quantity is less or equal to the quantity of the corresponding referenced product in stock. New orders may arrive, and there may be some arrivals of products in some quantity in the stock. We also consider that this company may decide to discontinue some products that have not been sold for some given time (e.g., six months). An order may be refused when the product is no more traded, or when the quantity ordered is not available in the stock and a decision was taken to discontinue the product; this refusal should take place within one month after the order was received. We take the hypotheses that the size of the company's warehouse is unlimited and that the traded products are not perishable.

The picture above shows how the invoice system matches the IS frame.

Due to lack of space, the complete specifications of the requirements and the the design part will not be given here but they are available in [6].

5.2 Domain and Requirements

As explained in Sect. 4, to specify the requirements in this case means to provide four specifications corresponding to the four parts of the frame, which are reported in the following subsections. Some specification modules are quite large, thus, for readability sake, we provide some "friendly" abbreviated presentation of them. The domain and requirements specifications share some common data structures, and by using the CASL construct for the declaration of named specifications, we have specified them apart and collected together under the title of basic data.

Basic Data Some obvious basic data are the codes for products, orders, and clients, and the quantities. We need also a notion of time encoded into a date (day/month/year). The components of an order are the date when it is received, the product ordered (referenced by its code), the quantity ordered, the client who issued the order (referenced by its code), and an order code. Moreover, to specify the invoice system, we need also to use the elaboration status of an order and the trading status of a product.

```
spec CODE =
    sorts Product_Code, Order_Code, Client_Code
    %% codes identifying the products, the orders and the clients

spec QUANTITY =
    sort  Quantity %% the quantities of the considered products
    ops 0 :→ Quantity;
        __+__ : Quantity×Quantity → Quantity, comm, assoc, unit 0;
        __−__ : Quantity×Quantity →? Quantity, unit 0 ;
    pred    __≤__ : Quantity×Quantity;
          ...

spec DATE =
    NAT then
free {
    type Date ::= __/__/__(Nat; Nat; Nat);   %% dates as day/month/year
    pred   __ ≤ __ : Date×Date;
    op  initial_date :→ Date;
        . . .

spec ORDER =
    CODE and QUANTITY and DATE then
    free type Order ::= mk_order(product : Product_Code; quantity : Quantity;
                                 date : Date;code : Order_Code; client : Client_Code)

spec STATUS =
    free types
        Product_Status ::= traded | not_traded;
        Order_Status ::= pending | invoiced | non_existing | refused;
    %% elaboration statuses of products and trading statuses of orders
```

Real World To specify the Real World component of the application domain, we have to express the relevant properties of its behaviour, following the schema introduced in Sect. 4.1; thus, we first determine the "events" and the "state observers", and then we look for the "incompatible events", the "relationships between state observers and events", and for the "event specific properties". We provide below the abbreviated presentation and a sketch of the corresponding CASL-LTL specification (see [6] for the full specification).

Events We present the events by listing the generators (written using capital letters) with their arguments and a short comment.

- *RECEIVE_ORD*(*Order*) to receive an order
- *SEND_INVOICE*(*Order*) to send the invoice for an order
- *REFUSE*(*Order*) to refuse an order
- *RECEIVE_PROD*(*Product_Code*; *Quantity*)
  to receive some quantity of a product
- *DISCONTINUE*(*Product_Code*) to discontinue a product
- *CHANGE*(*Date*) to change the date

State Observers We simply present the state observers by listing them with the types of their arguments and result, dropping the standard argument of the dynamic sort *State*. We use the notation convention that sort identifiers start with capital letters, whereas operation and predicate identifiers are written using only lower case letters.

- *product_status*(*Product_Code*) : *Product_Status* trading status of a product
- *order_status*(*Order_Code*) : *Order_Status* elaboration status of an order
- *available_quantity*(*Product_Code*) : *Quantity*
  available quantity of a product in the stock
- *date* : *Date* actual date

With the corresponding formal specification (see [6] for the full specification):

spec STATE_OBSERVERS =
    ORDER **and** STATUS **then**
    sort *State*
    ops $product\_status : State \times Product\_Code \to Product\_Status$
    %% trading status of a product
...

Incompatible Events We simply present the incompatible events by listing the incompatible pairs.

- All events referring to two orders with the same code are pairwise incompatible.
  - $RECEIVE\_ORD(o), SEND\_INVOICE(o')$ **s.t.** $code(o) = code(o')$
  - $RECEIVE\_ORD(o), REFUSE(o')$ **s.t.** $code(o) = code(o')$
  - $SEND\_INVOICE(o), REFUSE(o')$ **s.t.** $code(o) = code(o')$
  - $RECEIVE\_ORD(o), RECEIVE\_ORD(o')$ **s.t.**
    $code(o) = code(o') \wedge \neg (o = o')$
  - $SEND\_INVOICE(o), SEND\_INVOICE(o')$ **s.t.**
    $code(o) = code(o') \wedge \neg (o = o')$
  - $REFUSE(o), REFUSE(o')$ **s.t.** $code(o) = code(o') \wedge \neg (o = o')$
- All events referring to the same product are pairwise incompatible.
  - $RECEIVE\_PROD(p, q), SEND\_INVOICE(o)$ **s.t.** $product(o) = p$
  - $RECEIVE\_PROD(p, q), DISCONTINUE(p')$ **s.t.** $p = p'$
  - $SEND\_INVOICE(o), DISCONTINUE(p)$ **s.t.** $product(o) = p$
  - $RECEIVE\_PROD(p, q), RECEIVE\_PROD(p, q')$ **s.t.** $\neg q = q'$
- All change date events are pairwise incompatible.
  - $CHANGE(d), CHANGE(d')$ **s.t.** $\neg d = d'$

In the corresponding CASL specification (see [6]) each pair corresponds to an axiom, e.g., the first two axioms below.

$$st \xrightarrow{l} st' \wedge RECEIVE\_ORD(o) \in l \wedge SEND\_INVOICE(o') \in l \Rightarrow \\ \neg (code(o) = code(o'))$$

$$st \xrightarrow{l} st' \wedge RECEIVE\_ORD(o) \in l \wedge REFUSE(o') \in l \Rightarrow \\ \neg (code(o) = code(o'))$$

Relationships between State Observers and Events We simply present the relationships between state observers and events by listing for each state observer its initial value, which events modify it and how. This last part is given by stating which is the observer value after the happening of the various events. Notice that such value could be expressed by using also the observations on the state before the happening the event. Thus "**after** $RECEIVE\_PROD(p, q)$ **is** $available\_quantity(p) + q$" below means that the new value is the previous one incremented by q.

- $product\_status(p)$
  initially is *traded*
  after $DISCONTINUE(p)$ **is** *not_traded*
  not changed by other events
- $available\_quantity(p)$
  initially is 0

after $SEND\_INVOICE(o)$ **s.t.** $product(o) = p$
is $available\_quantity(p) - quantity(o)$
after $RECEIVE\_PROD(p, q)$ **is** $available\_quantity(p) + q$
not changed by other events

- $order\_status(oc)$
initially is $non\_existing$
after $RECEIVE\_ORD(o)$ **s.t.** $code(o) = oc$ **is** $pending$
after $SEND\_INVOICE(o)$ **s.t.** $code(o) = oc$ **is** $invoiced$
after $REFUSE(o)$ **s.t.** $code(o) = oc$ **is** $refused$
not changed by other events
- $date$
initially is $initial\_date$
after $CHANGE(d)$ **is** d
not changed by other events

Below, as an example, we report the complete axioms expressing the relationships between the state observer *product_status* and the events.

$$initial(st) \Rightarrow product\_status(st, p) = traded$$

$$st \xrightarrow{l} st' \wedge DISCONTINUE(p) \in l \Rightarrow product\_status(st', p) = not\_traded$$

$$st \xrightarrow{l} st' \wedge DISCONTINUE(p) \notin l \Rightarrow$$
$$product\_status(st', p) = product\_status(st, p)$$

Event Specific Properties We present the event specific properties by listing for each event the properties on the system state necessary to its happening (preconditions), the properties on the system state necessary after it took place (postconditions), and under which condition this event will surely happen. The system state before and after the happening of the event are denoted by st and st' respectively. It is recommended to provide as well for each event a comment summarizing its properties in a natural way. We give below the presentation of $RECEIVE\_ORD$.

$RECEIVE\_ORD(o)$
Comment: If the order o is received, then the product referred in o was traded, no order with the same code of o existed, the date of o was the actual date, and in any case eventually o will be either refused or invoiced.

before
$product\_status(st, product(o)) = traded$,
$order\_status(st, code(o)) = non\_existing$ **and** $date(o) = date(st)$
after
$order\_status(st', code(o)) = pending$ **and**
$in\_any\_case(st', eventually\ state\_cond(x \bullet$
$order\_status(x, code(o)) = refused \vee order\_status(x, code(o)) = invoiced))$

"$in\_any\_case(st', eventually\ state\_cond(x \bullet \ldots))$" is a formula of CASL-LTL built by using the temporal combinators. "$in\_any\_case(s, \pi)$" can be read "for every path σ starting in the state denoted by s, π holds on σ", where a path is a sequence of transitions having the form either (1) or (2) below:

(1) $s_0\ l_0\ s_1\ l_1\ s_2\ l_2\ \dots$ (infinite path)
(2) $s_0\ l_0\ s_1\ l_1\ s_2\ l_2\ \dots\ s_n$ $n \geq 0$

where for all i ($i \geq 0$), $s_i \xrightarrow{l_i} s_{i+1}$ and there does not exist l, s' such that $s_n \xrightarrow{l} s'$.

"*eventually state_cond*$(x \bullet F)$" holds on σ if there exists $0 \leq i$ s.t. F holds when x is evaluated by s_i.

Now we can give the formal specification of Real World for the invoice case, i.e., the company.

```
spec INVOICE_REAL_WORLD =
   ... then

   %% RECEIVE_ORD(o)
      st --l--> st' ∧ RECEIVE_ORD(o) ∈ l ⇒
      product_status(st, product(o)) = traded ∧
      order_status(st, code(o)) = non_existing ∧
      date(o) = date(st) ∧ order_status(st', code(o)) = pending ∧
      in_any_case(st', eventually state_cond(x •
              order_status(x, code(o)) = refused ∨
                 order_status(x, code(o)) = invoiced))
   ......
```

Information Requests We present the information requests by listing their generators with the types of their arguments; similarly for the information outputs.

- *available_quantity_of?(Product_Code)*
  what is the available quantity of a product in the stock?
- *quantity_of Product_Code sold_in Date – Date?*
  what is the quantity of a product sold in the period between two dates?
- *last_time_did Client_Code ordered?*
  what is the last time a client made some order?

Information Outputs

- *the_available_quantity_of Product_Code is Quantity*
- *error : prod_not_traded* the product appearing in the request is not traded
- *error : wrong_dates* the dates appearing in the request are wrong
- *the_quantity_of Product_Code sold_in Date – Date is Quantity*
- *Client_Code ordered_last_time_at Date*

Information Function Recall that the *inf_fun*, in this case named *invoice_inf_fun*, takes as arguments an information request and a history (a partial system lifecycle), defined as a sequence of transitions, i.e., precisely a sequence of states and labels. We simply present *invoice_inf_fun* by showing its results on all possible arguments case by case; each case is presented by starting with the keyword **on**, followed by the list of the arguments.

```
on $available\_quantity\_of?(p), h$
    if $product\_status(last(h), p) = traded$ returns
        $the\_available\_quantity\_of\ p\ is\ available\_quantity(last(h), p)$
    if $product\_status(last(h), p) = not\_traded$ returns $error : prod\_not\_traded$
on $quantity\_of\ p\ sold\_in\ d_1 - d_2?, h$
    if $\neg\ (d_1 \leq d_2$ and $d_2 \leq date(last(h))$ returns $error : wrong\_dates$
    if $d_1 \leq d_2$ and $d_2 \leq date(last(h))$ returns
        $the\_quantity\_of\ p\ sold\_in\ d_1 - d_2\ is\ sold\_aux(p, d_1, d_2, h)$
on $last\_time\_did\ cc\ ordered?,\ init(st)$ returns $initial\_date$
on $last\_time\_did\ cc\ ordered?,\ h\ l\ st$
    if $RECEIVE\_ORD(o) \in l$ and $client(o) = cc$ returns
        $cc\ ordered\_last\_time\_at\ date(st)$
    if $\neg\ (\exists\ o : Order \bullet SEND\_INVOICE(o) \in l$ and $client(o) = cc)$ returns
        $invoice\_inf\_fun(last\_time\_did\ cc\ ordered?, h)$
```

The auxiliary operation *sold_aux* returns the quantity of a product sold in a certain time interval, for its complete definition see [6].

As an example, we show below the complete CASL axioms corresponding to the definition of *invoice_inf_fun* for the first case.

```
$product\_status(last(h), p) = traded \Rightarrow$
    $invoice\_inf\_fun(available\_quantity\_of?(p), h) =$
    $the\_available\_quantity\_of\ p\ is\ available\_quantity(last(h), p)$
$product\_status(last(h), p) = not\_traded \Rightarrow$
    $invoice\_inf\_fun(available\_quantity\_of?(p), h) = error : prod\_not\_traded$
```

6 Conclusions and Future Work

While it is clear that methods are needed to help developing formal specifications, as extensively advocated in [4], this remains a difficult issue. This problem is addressed in [9,10] that define the concept of *agenda* used to provide a list of specification and validation tasks to be achieved, and apply it to develop specifications with Statecharts and Z. [14] also uses agendas addressing "mixed" systems (with both a static and a dynamic part), and provides some means to generate parts of the specification. [5] is the first work we know of that provides a formal characterisation of M. Jackson problem frames. Along this approach, we provide here a formalization of the translation frame and of the information system frame using the CASL language together with worked out case studies. Being in a formal framework lead us to add to the issues addressed by problem frames, the issue of correctness.

Following the approach proposed in this paper to use formal specifications in the development process of real case studies becomes an "engineering" kind of work. Indeed, for each frame we propose an operative method based on "normal" software engineering (shortly SE) concepts (inputs, outputs, events, ...) and not on mathematical/formal ones (existence of initial models, completeness, ...). Moreover, working with large case studies lead us to provide some legible presentations of the various parts of the specifications removing/"abstracting from" some conventional mathematical notations/overhead (while the corresponding complete specifications may be easily recovered from these) as for example, in Sect. 5.2.

We have based our work on some well established SE techniques and concepts (as the clear distinction supported by Jackson among the domain, the requirements and the design) that, for what we know, are not usually very well considered in the formal method community ([5] beeing an exception). Previous algebraic specifications of the case studies considered in this paper made by the authors themselves, without considering the SE aspects, were quite unprecise and perhaps also wrong. In the grep case everything was considered as "requirements" and then realized in the design phase, and so we had implemented also the regular expressions and the files. Instead, for the invoice case the old specifications were confused as regards what should be the responsibilities of system that we have to build (e.g., the information system was responsible to guarantee that an order eventually will be either invoiced or refused instead of simply taking note of when an order is invoiced).

Let us note that, while the selection of the correct frame and the specification of the requirements and the design are essential, specifying the domain part is necessary to produce sensible requirements, and may also be needed to discuss with the clients, or to check about possible misunderstandings with the domain experts (most of the worst errors in developing software systems are due to wrong ideas about the domain).

Another relevant aspect of our work is clearly "reuse": but here we reuse what can be called, by using a current SE terminology, "some best practices", not just some specifications. The ways to handle particular problem frames that we propose encompass the practice on the use of algebraic specifications of the authors; and so our work may be considered in the same line of the use of "patterns" ([8]) for the production of object oriented software. The most relevant difference between [8] and the work presented here is the scale: we consider as a reusable unit a way to solve a class of problem, the patterns of [8] consider, instead, something of much smaller (pieces of the design).

Acknowledgements We would like to thank the anonymous referees for their careful reading and helpful comments.

References

1. M. Allemand, C. Attiogbe, and H. Habrias, editors. *Proc. of Int. Workshop "Comparing Specification Techniques: What Questions Are Prompted by Ones Particular Method of Specification". March 1998, Nantes (France).* IRIN - Universite de Nantes, 1998.
2. E. Astesiano, B. Krieg-Bruckner, and H.-J. Kreowski, editors. *IFIP WG 1.3 Book on Algebraic Foundations of System Specification.* Springer Verlag, 1999.

3. E. Astesiano and G. Reggio. Labelled Transition Logic: An Outline. Technical Report DISI–TR–96–20, DISI – Università di Genova, Italy, 1996.
4. E. Astesiano and G. Reggio. Formalism and Method. *T.C.S.*, 236, 2000.
5. D. Bjørner, S. Kousoube, R. Noussi, and G. Satchok. Michael Jackson's Problem Frames: Towards Methodological Principles of Selecting and Applying Formal Software Development Techniques and Tools. In M.G. Hinchey and Liu ShaoYing, editors, *Proc. Intl.Conf. on Formal Engineering Methods, Hiroshima, Japan, 12-14 Nov.1997.* IEEE CS Press, 1997.
6. C. Choppy and G. Reggio. Using CASL to Specify the Requirements and the Design: A Problem Specific Approach – Complete Version. Technical Report DISI-TR-99-33, DISI – Università di Genova, Italy, 1999. `ftp://ftp.disi.unige.it/person/ReggioG/ChoppyReggio99a.ps`.
7. G. Costa and G. Reggio. Specification of Abstract Dynamic Data Types: A Temporal Logic Approach. *T.C.S.*, 173(2), 1997.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
9. W. Grieskamp, M. Heisel, and H. Dörr. Specifying Safety-Critical Embedded Systems with Statecharts and Z: An Agenda for Cyclic Software Components. In E. Astesiano, editor, *Proc. FASE'98*, number 1382 in LNCS. Springer Verlag, Berlin, 1998.
10. M. Heisel. Agendas – A Concept to Guide Software Development Activities. In R. N. Horspool, editor, *Proceedings Systems Implementation 2000.* Chapman & Hall, 1998.
11. M. Jackson. *Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices.* Addison-Wesley, 1995.
12. P.D. Mosses. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In M. Bidoit and M. Dauchet, editors, *Proc. TAPSOFT '97*, number 1214 in LNCS, Berlin, 1997. Springer Verlag.
13. The CoFI Task Group on Language Design. CASL The Common Algebraic Specification Language Summary. Version 1.0. Technical report, 1999. Available on `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/`.
14. P. Poizat, C.Choppy, and J.-C. Royer. From Informal Requirements to COOP: a Concurrent Automata Approach. In J.M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems*, number 1709 in LNCS. Springer Verlag, Berlin, 1999.
15. G. Reggio, E. Astesiano, and C. Choppy. CASL-LTL: A CASL Extension for Dynamic Reactive Systems – Summary. Technical Report DISI-TR-99-34, DISI – Università di Genova, Italy, 1999. `ftp://ftp.disi.unige.it/person/ReggioG/ReggioEtAl99a.ps`.
16. M. Roggenbach and T. Mossakovski. Basic Data Types in CASL. CoFI Note L-12. Technical report, 1999. `http://www.brics.dk/Projects/CoFI/Notes/L-12/` .

JTN: A Java-Targeted Graphic Formal Notation for Reactive and Concurrent Systems *

Eva Coscia and Gianna Reggio

DISI, Università di Genova – ITALY
e-mail: {coscia,reggio}@disi.unige.it – fax: 39-010-3536699

Abstract. JTN is a formal graphic notation for Java-targeted design specifications, that are specifications of systems that will be implemented using Java.
JTN is aimed to be a part of a more articulated project for the production of a development method for reactive/concurrent/distributed systems. The starting point of this project is an existing general method that however does not cover the coding phase of the development process. Such approach provides formal graphic specifications for the system design that are too abstract to be transformed into Java code in just one step, or at least, the transformation is really hard and complex.
We introduce in the development process an intermediate step that transforms the above abstract specifications into JTN specifications, for which the transformation into a Java program is almost automatic and can be guaranteed correct. In this paper we present JTN on a simple toy example.

Introduction

In this paper we present a part of a more articulated project we are currently working on: a development method for reactive/concurrent/distributed systems (shortly systems from now on) that are finally implemented in Java. Such development process should be supported by formal tools and techniques, whenever possible, and by a set of user guidelines that describe in detail how to perform the various tasks. The formal bases and the main ideas come from previous work of one of the authors about the use of formal techniques in the development of systems that, however, did not ever considered the final coding step, see, e.g., [1, 2, 10, 11]. We chose Java as the implementation language since it is OO, widely accepted for its simplicity and, at the same time, for its richness. It is considered a language for the net, for its portability, but also a language for concurrency and distribution. Moreover, there exists a precise, even if informal, reference [4] for the semantics of the core language.

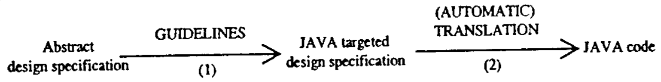
[11] presents a general method for giving formal graphic design specifications of systems, but such specifications are too abstract to be transformed into Java code in just one step, or at least, the transformation is really hard and complex.

* Partially funded by the MURST project: Sistemi formali per la specifica, l'analisi, la verifica, la sintesi e la trasformazione di sistemi software.

For example, following [11] you can specify systems with n-ary synchronous communications, where the components can exhibit any kind of non-determinism and can be coordinated by complex scheduling policies, which cannot have a direct implementation into Java.

Moreover, the complexity of this transformation into Java does not allow to check the correctness of the generated code, and there is no way to automatize it. Furthermore, [11] does not take into account the relevant, good characteristics of Java, as the OO features.

We think that it is useful to introduce an intermediate step in the development process that transforms an abstract design specification into a Java-targeted one, whose transformation into a Java program is really easy.


Abstract design specification
GUIDELINES
(1)
JAVA targeted design specification
(AUTOMATIC) TRANSLATION
(2)
JAVA code

Step (2) can be automatized and guaranteed correct whereas step (1) cannot be automatized, but we are working to give a rich set of guidelines for helping the user in such task.

Here we present JTN, a graphic formal notation for the Java targeted design specifications, obtained by targeting [11] to Java, that is by modifying the specification language to take into account the features and the limitations of Java.

JTN is graphic because every aspect of system (e.g., components, global architecture and behaviour) is described only by diagrams. But it is formal, because the diagrams composing the specifications are just an alternative notation for logic specifications as in [11] (the formal semantics of JTN is presented in [3]).

We think that JTN, with the associated method and software support tools, could help the development of reliable systems implemented in Java.

- We can describe the system design graphically, and that is helpful to grasp the system characteristics. However, the JTN graphic specifications are structured and that avoids one of the possible drawbacks of graphic notations: to handle very large diagrams that could be not understood.
- The level of the JTN descriptions is not too low; the designer avoids to specify too much details and the drawings are simple enough; for example, in JTN there are user defined data-types and abstract communication mechanisms, as synchronous and asynchronous channels.
- JTN specifications are formal, with the usual advantages to use formal methods without bothering the specifiers with too much formalities.
- The automatizable translation to Java reduces the time to a working system and also gives a prototyper for such specifications.

There are neither theoretical nor practical problems to realize a full tool-set for supporting the use of JTN using the current technology (e.g., interactive editor, static checker, hyper-textual browser, translator to Java, debugger); it is possible to realize them within a reasonable amount of time, we just need some human resources.

Here, for lack of room, we consider only a rich subset of JTN applied on a toy running example and give some ideas about its translation into Java; a detailed presentation of JTN and further examples are in [3]. In Sect. 6 we present the relations with other works as well as some hints on our future work.

The running example We specify the design of a Java program simulating a pocket calculator that computes and interacts with a keyboard, a display and a printer; think, for example, of a small application simulating a calculator on the desktop of a computer. The functionalities of the calculator are quite obvious: it can receive, by the keyboard, numbers and simple commands for performing operations (addition and multiplication) and for printing the display content.

1 JTN

In this section we first describe the main features of the abstract design specification technique of [11]; then we describe how to target it to Java and give an overall presentation of the JTN notation.

1.1 Abstract design specifications

The specification technique of [11] distinguishes among the data-types, the passive and the active components of a system, because these components have a different conceptual nature and play a different role within the systems.

The *data-types* are static structures used by the other components, with no idea of an internal state changing over time. The *passive* components have an internal state that can be modified by the actions of other active components. The *active* components have an internal state, but they are also able to act on their own (possibly by interacting among them and with the passive components).

We can completely describe a data-type by giving its values and the operations over them, whereas we describe the passive components by giving their states and the actions that can be performed over them, which obviously can change such states. We want to remark this difference. Data-types define stateless elements (essentially, values) that are used by (active and passive) components. Instead, the passive components are actual components of the system, having an internal state that can be updated by the active components. Finally we describe the active components by giving the relevant intermediate states of their lives and their behaviours, which are the possible transitions leading from one state to another one. Every transition is decorated by a label abstractly describing the information exchanged with the external, w.r.t. to the component, world. Notice that many transitions can have the same source, and that allows to handle nondeterminism.

Let us consider, for simplicity, a system having a database inside. The database is a passive component whose internal state is modified by the operations it supplies outside; the data managed and exchanged by the database are data-types; and the processes using the database are the active components of the system.

The activity of a system results by describing how its components *cooperate*, i.e., to say which transitions of the active components and which operations over the passive components have to be performed together, and which is the exchange of information with the external (w.r.t. the whole system) world.

A system, in turn, can be seen as an active component of another system, and so we can specify systems with a multi-level architecture.

The underling formal model for an active component (and thus for a system) is a labelled transition system (LTS), that is triple consisting of a set S of elements (intermediate states), a set L of the labels (interactions with the external world) and a ternary transition relation, a subset of $S \times L \times S$ (transitions). The passive components and the data-types are modelled by first-order structures.

The graphic specifications of [11] follow the system structure; thus they consist of diagrams for the data-types, for the components (the behaviour of the active components is represented as a kind of finite automata) and for the cooperation among them. Formally, these diagrams correspond to an algebraic/logic specification having LTS's as models, see [11]. Note that the specification language of [11] has neither concepts nor mechanisms related to OO, nor features of some particular programming language, as handshake communications and asynchronous channels; instead it allows the specifier to directly define any feature of the system by describing the corresponding behaviours and cooperation.

1.2 Java-Targeting

We designed JTN by adapting the technique presented in Sect. 1.1 to the features and to the limitations of Java.

We want to keep distinct, at this more concrete level too, the concepts of data-types, passive and active components of a system. In our opinion, it is useful to have this distinction to avoid confusion and to make the specification more readable.

Then, we introduce the new OO concepts of class and instance and provide an explicit representation of the relevant relationships among them, as inheritance and use; but in JTN we have three kinds of classes, one for each kind of entities that we consider: data-types, passive and active components.

The data-types are described at an abstract level by giving their constructors and by defining the associated operations, without considering an OO perspective; however it is easy to transform them into Java classes.

The passive components are seen as objects, whose state is given by a set of typed fields and the operations to modify them are methods. The transformation of such object specifications into Java classes is immediate.

The active components are seen as processes, with a state, an independent activity and communication channels to interact with other active components and with the external world. In this case, the natural implementation is given by Java threads. We chose to use communication among processes via channels, rather than via method calls. In this way, a process does not offer methods outside and we do not have to manage method calls while the process is performing its activity (in Java, there is no built-in mechanism to disable method calls).

As Java objects and threads communicate by method calls and streams, the main typologies of cooperation are: between a process and an object, by means of a call to an object method, and among a set of processes by communication along asynchronous and synchronous channels. The asynchronous channels are rendered by streams, and the synchronous ones are implemented by particular additional objects.

Java supports only system architectures of at most two levels. The first level corresponds to multi-threaded Java programs, and the second corresponds to distributed Java applications consisting of programs possibly running on different machines. For lack of room, in this paper we do not consider the second level.

For the same reason, here we consider simple objects and processes, i.e., without sub-objects and sub-processes, so calls to methods of other objects cannot appear in a method body. Moreover, we do not consider the dynamic creation of process and objects: we assume to start with an initial configuration of the system where all the components have been already created in the initial state.

Using the JTN concepts we model the running example as follows. The calculator has four active components: the keyboard driver reading keys from the keyboard, the computing unit performing the computations, the display driver echoing inputs and results to the display, and the print driver printing the display content. All such processes use an object, which records the content of the display, and three different types of data: digits, lists of digits and commands.

The keyboard driver receives the keys by an asynchronous channel from the keyboard (the external world); the display driver and the print driver send their outputs to, respectively, the display and the printer by two other asynchronous channels. The communications and the synchronizations among the active components are realized by some synchronous channels.

1.3 Overall structure of the JTN specifications

We factorize a system specification into several diagrams showing different aspects or parts of the system. Thus the diagrams are not too large and complicated, and so really useful. For example, some diagrams focus on the behaviour of the components and other focus on the architectural structure of the system.

Class Diagram The *class diagram* captures the classes of the system components and their relationships. We consider three different kinds of classes: *data-types*, *object classes* (for passive components) and *process classes* (for active components), graphically represented by different icons.

All the information about a class is given by two complementary diagrams: *interface* and *body*. The first one describes which are the services (different for each kind of class) that the class offers outside. The latter defines such services and can be given apart from the class diagram. The forms of the two diagrams depend on the class kind and are described in deeper details in Sect. 2.

In an OO perspective, stand-alone classes are not so meaningful; most of them are related to accomplish more complex functionalities. Thus, we complete the class descriptions with the relevant relationships among them.

Inheritance between classes of the same kind, it states that a class is a specialization or an extension of another one.

Usage states that a class (of any kind) uses the data defined by a data-type.

Clientship states that a process class assumes the existence of an object class, as it can use its methods.

JTN defines the operations, the methods and the process behaviours inside the body diagrams by ordered lists of conditional rules, with a uniform presentation, just a general form of "guarded commands". The alternatives of a command are evaluated in order and the first one having a satisfied guard is chosen. The guards are partly realized by a boolean condition and partly by pattern matching (as in ML [6]) over the parameters of the operation, of the method call and over the state of the process respectively. The use of pattern matching is useful to make shorter and more readable the whole definitions. Let us remark that we avoid problems with overlapping patterns and conditions by explicitly ordering the guards.

Architecture Diagram The architecture diagram describes which are the components of the system, and how they interact (by which communication channels and by which method calls).

Sequence Diagram A sequence diagram is a particular form of message sequence chart (see [8]) that describes a sequence of actions occurring in a (possibly partial) execution of the system and involving some components. The represented actions are communications over channels and method calls. We introduced these diagrams because they are used in the most widely accepted specification techniques in the field of Software Engineering, such as UML.

The class diagram (with the possibly separated body diagrams) and the architecture diagram fully describe a system. The sequence diagrams are an additional way to present information on the system that is very intuitive and easy to be understood.

2 Class Diagram

There is one global class diagram for the whole system, representing the classes of all its constituents. It is a graph, where the nodes represent classes and the arcs class relationships. The icons for a data-type, an object class and a process class are, respectively, ▭ ▭ ▭.

For each class there are two diagrams, *interface* and *body*, both with a slot with the name of the class, i.e., the type of its elements. The interface diagrams are always in the class diagram, whereas the body ones can be given separately.

The contents of the interface and of the body diagrams vary with the kind of the class and in the following subsections we present them and the relationships among classes. In fig. 1 we report the class diagram for the calculator example.

Data-type We chose to describe a data-type by giving its constructors and defining the associated operations by means of conditional rules with pattern matching a la ML (see [6]).

The interface diagram for a data-type contains the list of its visible constructors and operations, and the body diagram contains the the private constructors and the definition of the visible and private operations. The body diagram is divided into many slots, separated by dashed lines, each one containing the definition of an operation, by conditional rules.

The most common data-types, either basic (e.g., NAT) or parametric (e.g., LIST), are predefined and implicitly used by all the classes, so we do not report them in the class diagram. Moreover, data-types defined by combinations of predefined ones can be renamed and grouped together. In fig. 1 DIGIT is a renamed subrange of CHAR and KEY is the union of DIGIT and COMMAND.

The APPLY data-type, defined in fig. 1 by inheriting from the others, contains some operation definitions, one public, Apply, and two private, Code and Decode. It implicitly uses NAT. Decode is defined by using the pattern matching: given an actual parameter a, if a matches the pattern Empty (i.e., $a=$ Empty, Empty is a constant constructor), then it returns 0; if a matches d::dl (i.e., $a = e :: l$, :: is the list constructor adding an element to a list), then it returns (Ord(e)-Ord('0')) + 10 * Decode(l).

Object Class An object is a passive component of the system, which has an internal state but it does not perform an independent activity. Objects cooperate with other components by offering services (i.e., methods) that the processes can call to complete complex functionalities.

Here, for lack of room, we do not present the complete version of the object classes with sub-objects and local methods.

Interface Diagram The interface diagram of an object class contains the list of the public methods with their names, the types of their parameters and of the returned values (if any). The DISPLAY class (fig. 1) has two methods, Write and Add, with one parameter of type DIGIT_LIST, and Read, with no parameter, that returns a DIGIT_LIST value.

Body Diagram The object body diagram is divided into two slots, containing:

- the *fields*; in JTN there are only private fields that can be accessed only by methods. For each field we give the name and the type plus its initial value (see field Cont in fig. 2);
- the definition of *public* and *private* methods.

Using JTN, we define the methods by conditional rules with pattern matching.

A method $\mathrm{M}(\mathrm{IT}_1, \ldots, \mathrm{IT}_k)$: OT is defined by an ordered list of conditional rules whose form is

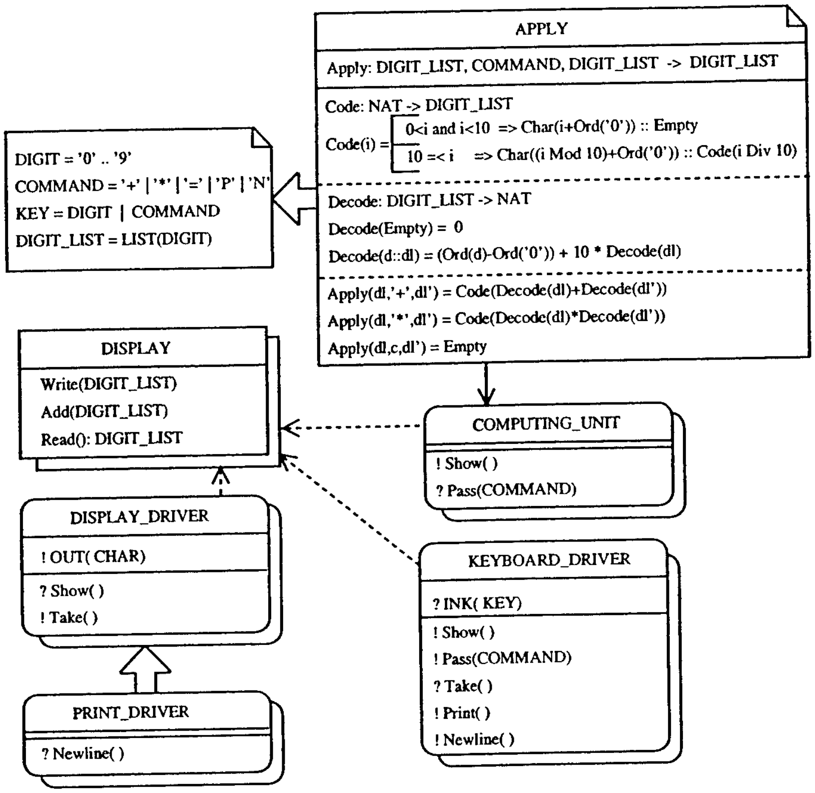


Fig. 1. Calculator: Class Diagram

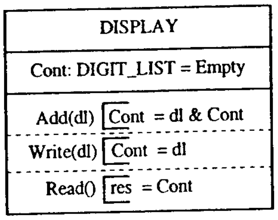


Fig. 2. Calculator: DISPLAY body diagram

$$M(i_1,\ldots,i_k)\left|\begin{array}{l}\text{if cond}_1\ \{\ \text{assignement\_list}_1\ \}\\ \hline \ldots \\ \hline \text{if cond}_n\ \{\ \text{assignement\_list}_n\ \}\end{array}\right.$$

where: for each h ($1 \leq h \leq k$) i_h is a pattern, i.e., an expression of type IT_h built only by constructors and variables; for each j ($1 \leq j \leq n$) $cond_j$ is a boolean expression over the variables in $i_1,..,i_k$ and the object fields; $assignment\_list_j$ is a list of assignments of form either "f = e" or "res = e", with e an expression over the variables in $i_1,..,i_k$ and the object fields, f a field name and res a special variable of type OT denoting the result returned by the method.

A method call is executed as follows. We find the first, w.r.t. the ordering, rule whose pattern matches with the parameters of the call and with a condition that holds. If none matches, then we have an error. Then, the corresponding assignments are performed. The value returned by the method (if any) is the final value of the variable res. Trivial examples of method definitions are in the body of the class DISPLAY in fig. 2 (& is the operation for appending lists).

Process Class In our approach, processes are different from objects and their description cannot be given in the same way. First note that processes are *active* components that behave independently and do not offer methods outside. Their behaviours are not sequential, instead they run concurrently and cooperate by message exchange with those of the other processes. So, the process interface diagram does not contain methods, but the communication channels, synchronous and asynchronous.

The body diagram describes the behaviour of the process, by presenting its interesting intermediate states, each one characterized by a name and typed parameters, and its transitions, precisely from every intermediate state, some conditional rules define all the states it can reach by interacting with the external world (i.e., the labelled transitions of [11]). In the general method of [11], there is no restriction on the form of the external interactions, which are just described by labels. In JTN, a process can communicate with the external world only by calling the object methods or by using the communication channels.

We give a graphic presentation of the behaviour that naturally depicts what a process does, by showing all the possible transitions starting from any state.

Interface diagram JTN processes use two kinds of channels: synchronous and asynchronous; both kinds of channels are distinguished into input and output ones. Thus the interface diagram for a process class has two slots containing the *asynchronous* and the *synchronous* channels, respectively, with their names, their directions (described by ! and ?) and the types of exchanged values (if any).

For example, the DISPLAY_DRIVER interface diagram in fig. 1 declares two synchronous channels Show and Take used only for a synchronization purpose (no value exchanged), and an asynchronous one, OUT, on which an instance of DISPLAY_DRIVER sends a char outside.

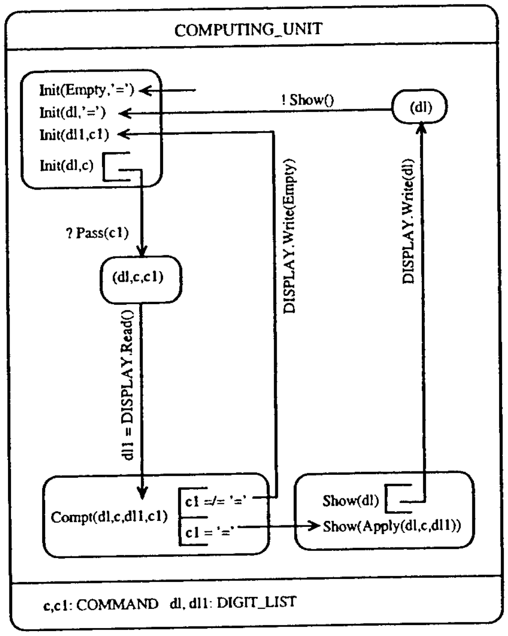


Fig. 3. Calculator: Computing Unit behaviour graph

Behaviour Graph [1]*(process body diagram)*
The *behaviour* of a process of a class is described by a graph, whose arcs represent "generic" labelled transitions, whose form is

$$S(pt_1,\ldots,pt_n)\ \lfloor cond(pt_1,\ldots,pt_n) \xrightarrow{l} S'(e_1,\ldots,e_k)$$

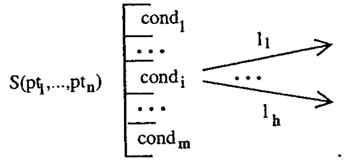
where S, S' are state constructors, pt_1, ...,pt_n are patterns for the state parameters, l is a pattern for the external interaction and e_1,..,e_k are expressions over the variables in l and $pt_1,\ldots,pt_n$.

The label l can have six different forms, depending on the kind of interaction that is performed with the external world:

- ?ACH(x), !ACH(e): input from/output to an asynchronous channel;
- r = oe.m(x_1, ..., x_n) (or oe.m(x_1, ..., x_n) if m has no return type): object method invocation;
- ?sch($x_1,\ldots,x_k$),!sch($e_1,\ldots,e_k$): input from/output to a synchronous channel;
- τ: internal activity, usually omitted.

[1] The behaviour graphs of JTN play the same role of the UML state diagrams.

All the sources and targets of transition representing state patterns of the same kind are grouped together into a node of the behaviour graph, avoiding repetitions. Thus each state pattern is written once, and the conditions are listed aside the corresponding state pattern to form an *alternative*, as below



The arrows leaving a condition are ordered. Thus, inside a node of the graph we have an ordered list of alternatives plus state patterns that are only targets of transitions.

The interpretation of the behaviour graph is as follows. When a process is in a state K(args), we consider in order all the alternatives inside the node for the K states, until we find one whose pattern matches args. Then, inside the chosen alternative, we look for the first true condition. Finally we consider the labels on the arrows leaving the condition, trying to determine the first one that can be executed (recall they are ordered). If no matching pattern with a true condition is found, then the process is definitively stopped.

Not all choices of labels (l_1,..,l_h) are meaningful; the admissible cases are as follows, and for each of them we explain how to select the one to execute:

1. $h = 1$ and l_1 is a method call or an output on an asynchronous channel or an internal transition; the corresponding transition can be executed.
2. $h \geq 1$ and for each i ($1 \leq i \leq h$) l_i = !sch_i(...) or l_i = ?sch_i(...): all synchronous channels sch_i are checked in the order; if the communication on sch_i cannot be executed, then the following one is checked. If no communication can be executed, the process is suspended until the last communication completes.
3. $h \geq 1$ and for each i ($1 \leq i \leq h$) l_i = ?ACH_i(...): the asynchronous channels ACH_i are continuously tested in the order, until an available message is found.

In cases 2 and 3 we can add an arrow labelled with “else” with the meaning that whenever no other transition can be executed, such escape will be performed as an internal action, leading to another state from which the activity continues. An else label can be used in case 2 when no communication is immediately available, to return to the same state and start again the polling procedure. See e.g., the state Taking in fig. 4, where KEYBOARD_DRIVER can perform a synchronous communication on channel Take; otherwise KEYBOARD_DRIVER moves to another state (by else transition) in which it tries to read a character from the asynchronous input channel INK, and, if nothing is available, by another else transition, then it will come back to the state Taking.

Instead of explicitly declaring in a behaviour graph each state constructor with the type of its parameters, we add a slot for declaring the types of the used variables; obviously each state constructor must be typed consistently.

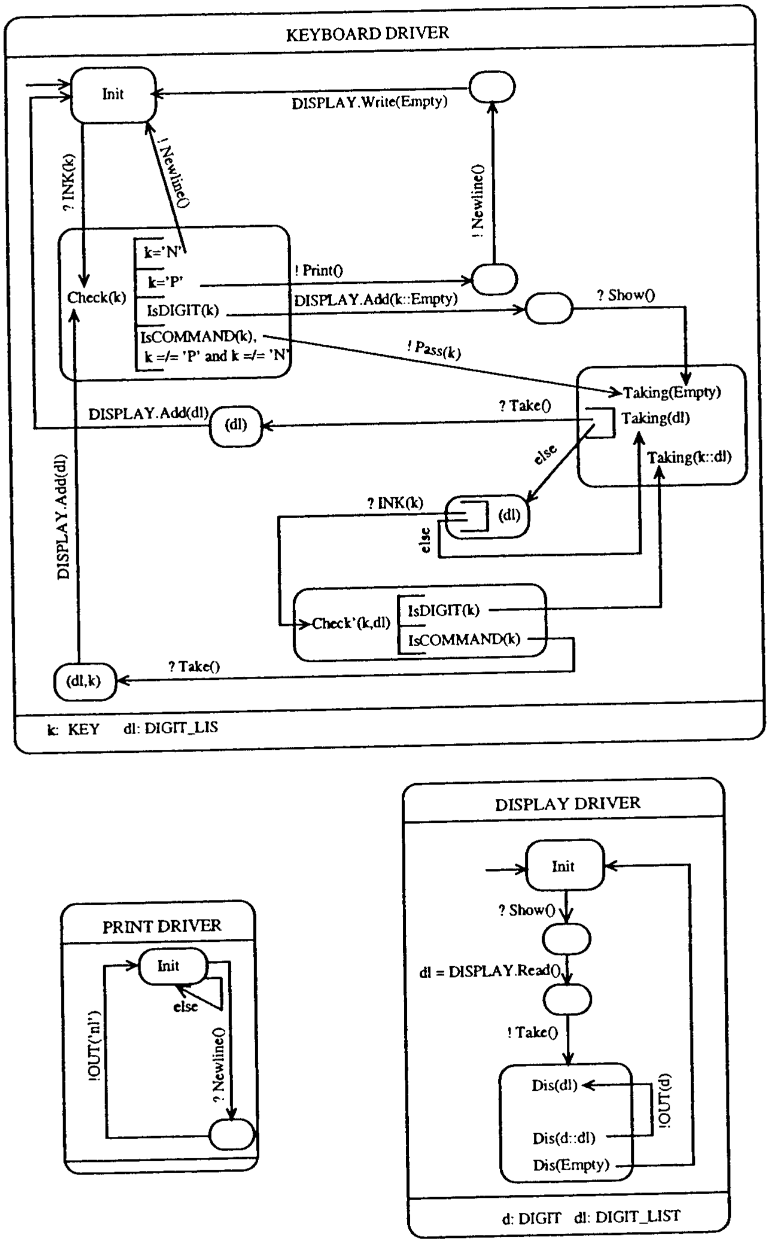


Fig. 4. Behaviour graphs

In fig. 3 and 4 we omit the name of a state every time it is not relevant; in such a case, the icon is empty, or just contains the list of the arguments.

An arrow with neither starting state nor label enters in the initial state of the system (see the upper left arrow in fig. 3)

Class Relationships Here we briefly illustrate the relationships among classes that we can put in the class diagram.
Inheritance (⇦) states that a class extends another one. It is restricted to classes of the same kind. What really the word "extends" does mean, depends on the particular kind of class. With regard to data-types, inheritance is used to add new operations. An example is APPLY, that adds the operation Apply. When considering an object class, inheritance is a mechanism for adding new methods and fields; finally, when considering a process class, inheritance adds transitions (i.e., behaviour) and new communication channels.

In our example, the PRINT_DRIVER class inherits from DISPLAY_DRIVER: its interface diagram is the one of DISPLAY_DRIVER with a new synchronous channel Newline; the behaviour graph of class PRINT_DRIVER depicts only the new transitions (see fig. 4), implicitly assuming those described in the behaviour graph of DISPLAY_DRIVER. More precisely, the transitions starting from states of the same kind are merged together; the new alternatives for a given state, as well as new transitions associated with an existing condition, are added at the end of the list, as they represent alternatives to be considered after the existing ones (we are currently studying more suitable mechanisms to describe how to re-order these alternatives).

The three different inheritance relations define three hierarchies over, respectively, data, object and process types (i.e., classes).
Usage (⟵) states that a data-type is used by another class. It is represented by an arrow from the used data-type class to the using class. If all the system components use a data-type, then the usage relation is omitted.
Clientship (⟵) states that a class assume the existence of another one, because it calls its methods. In our example, the process of all classes call the methods of the DISPLAY class.

In this paper we do not consider structured objects, calling other object methods, so clientship relates only process classes with the classes of the objects whose methods they call.

3 Architecture diagram

The architecture diagram describes the structure of the system showing its components and how they cooperate. The icons for the process and the object instances are slightly different from the corresponding ones for the classes; they are single boxes or single boxes with rounded corners: □ ▢.

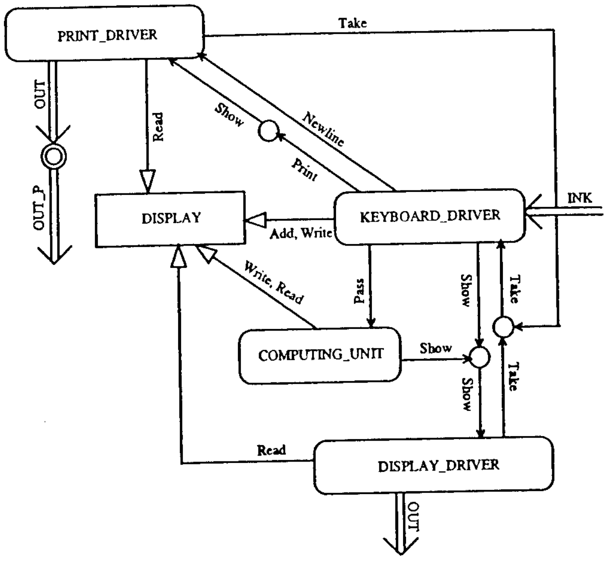


Fig. 5. Calculator: Architecture diagram

An instance icon contains only the instance identifier with the name of the corresponding class, separated by colons; the identifier is omitted if there is only one instance for such class, as in fig. 5.

The architecture diagram is a hyper-graph whose nodes are class instances that represent the components, and whose hyper-arcs represent how they cooperate. Let us remark that in this work we consider neither creation/deletion of components nor architectures having a generic number of components.

We can distinguish three kinds of hyper-arcs, representing:

method call: a process calls a method of an object; the icons of the two instances are linked by an arrow decorated with the method name; the arrow is oriented from the caller process to the called object.

asynchronous communication: fig. 6a) describes the connection of some asynchronous channels; OAC_i are output channels and IAC_i are input channels of the processes attached to the hyper-arc. The type of the exchanged message is the same for all the channels. Moreover, the channel types and versus must be in accord with the interfaces of the classes of the connected processes.

We can distinguish some cases. If $n = 1, m = 0$ or $n = 0, m = 1$, then the icon describes a channel for a process that communicates with the external world (e.g., the OUT channel associated with the DISPLAY_DRIVER in fig. 4). If $n > 0, m > 0$, a message sent on a generic OAC_i will be replied on all IAC_1,..,IAC_m. If $n = 1, m = 1$ the channel connects two processes or it is used to rename a channel, as we can see in fig. 5, where channel OUT of PRINT_DRIVER is renamed as OUT_P to avoid name clash with the same channel of DISPLAY_DRIVER.

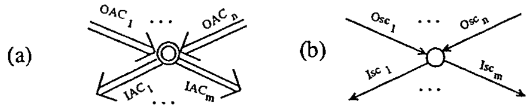


Fig. 6. Connectors for asynchronous and synchronous channels

synchronous communication: fig. 6b) describes the connection of some synchronous channels. We always have $n > 0, m > 0$, because synchronous channels cannot be used for communication with the external world. Again, the channel types and versus must be in accord with the interfaces of the classes of the connected processes.

The synchronous communication always involves two processes at a time: one process connected on a generic Osc_i acting as a sender, and one connected on a generic Isc_i acting as a receiver. Thus, the drawing can be interpreted as a shortcut for a set of channels connecting pairwise all sending to all receiving processes. An example is channel Show in fig. 5 that connects KEYBOARD_DRIVER, COMPUTING_UNIT (senders) and DISPLAY_DRIVER (receiver).

4 Sequence Diagram

A sequence[2] diagram is a kind of Message Sequence Chart [8] that gives a (possibly partial) description of a (possibly partial) execution of the system. Sequence diagrams are of particular interest because introduce in a specification formalism a technique that is used in the most widely accepted methods and notations in the Software Engineering field (such as UML).

A sequence diagram graphically represents some components taking part in a partial system execution and the ordered sequence of interactions among them and with the external world performed during such execution. The considered interactions are communications over channels and calls to object methods. The graphic presentation enlightens relevant aspects of the temporal ordering among interaction occurrences. Moreover, the diagram can be annotated with information about the state of the components, so it is possible to represent effects or conditions of action occurrences on single components.

The class and the architecture diagram supply complementary information about the system, whereas the sequence diagrams are just a different way to visualize information that has been already specified by the other diagrams. Several sequence diagrams may be presented for the same specification, to cover, e.g., the description of some interesting use cases of the system.

Sequence diagrams are not valuable for their information content (because it is already present in the other diagrams) but mainly from a methodological point of view and can be used for different purposes, for instance:

[2] In this case we use the same terminology of UML, since our sequences and the UML ones are rather similar; we can analogously define a form of collaboration diagram.

- to give a more natural and clear representation of the developed system to a client (e.g., to show how the calculator performs the addition);
- to show that the behaviour of the system specified by the class and the architecture diagrams is, in particular circumstances, the expected one. For example, by a sequence diagram we can show that if the user does not digit an "=" at the end of the operation, then the calculator does not return the result. From our experience, the construction of sequence diagrams, also if "by hand", helps to control the quality of the proposed design and allows to detect errors and omissions in the specification.

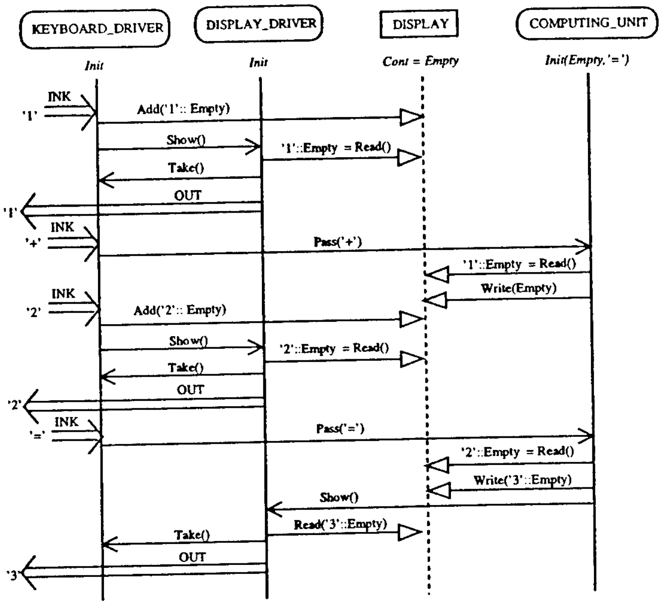


Fig. 7. Calculator: a sequence diagram for the computation of 1 + 2

Sequences diagrams are forms of message sequence charts, thus there are *vertical lines* representing the lives of the components involved in the execution. We use a dashed line to represent objects and a continuous line to represent processes. The *horizontal lines* describe the interactions occurring among the components or with the external world (i.e., communications on a channel, and method calls). Lines are put from top to bottom with respect to the temporal ordering of happening.

We use different icons for asynchronous communication (a double arrow), synchronous communication (a single arrow) and method call (a single arrow with outlined head) as we can see in fig. 7.

An asynchronous communication with the external world is just an incoming or outcoming double arrow, labelled by the channel name and by the exchanged data. An asynchronous communication between two processes is represented by two broken arrows. The part representing the start of the communication (send) is over the other one. They are separated by vertical dots and the data exchanged is annotated over both the two parts. Other actions may occur between the two phases of the asynchronous communications. A synchronous communication is decorated by the name of the channel and by the exchanged message. A method call is decorated by the name of the method and the parameters.

At any point of the vertical lines it is possible to put conditions on the value of the fields, for an object, and on the state and its arguments, for a process. The starting state of the execution may be described by such annotations, on the top of the corresponding vertical lines (see fig. 7).

Note that the elimination of a component and of its interactions returns another sequence diagram. If we drop DISPLAY_DRIVER in fig. 7, then we have a sequence diagram concerning only the updating of DISPLAY.

As sequence diagrams can erroneously depict executions that are not coherent with the rest of the specification, we must define when a sequence diagram is consistent with the information supplied by the class and the architecture diagrams.

Once we fixed the starting state of each instance, we can easily trace out how the system evolves. The object body diagrams describe how a method execution changes the state of an object. The behaviour graphs describe which communications or which method calls a process can perform from a given state and thus is the corresponding new state. The architecture diagram presents the topology of both the external as well as the inter-process communications. Thus, when we know which are the values arriving on the input asynchronous channels from the external world, we can find which actions the system components can perform and consequently which states the system can reach.

So, given a sequence diagram, we can determine the starting state of the system and then whether the depicted interactions can happen in the depicted order.

This consistency idea can be precisely defined, remembering that JTN is a formal specification language (the semantics of class plus architecture diagrams is an LTS) and that each sequence diagram corresponds to a formula in a branching time temporal logic saying that, from the starting state there exists a sequence of transitions where the depicted communications happened in the depicted order.

5 Implementation of main mechanisms

Here we briefly sketch out the implementation in Java of some of the JTN mechanisms. Obviously, the resulting Java program manages the classes and the instances shown in the diagrams, and also some auxiliary ones that are the standard implementation for synchronous and asynchronous channels and predefined data-types.

The predefined data-types are mostly obtained by combining Java primitive data (e.g., integer) and by extending some Java standard classes (e.g., Vector). The user-defined data-types have a standard translation: the constructors and their arguments are implemented as instance private fields; the component extractors operating on the data are trivial methods returning the value of the corresponding fields. The operations are translated into methods, whose code implements the guarded commands and the pattern matching used to define them.

The object classes are implemented as Java classes. The private fields implement the fields, initialized to the value represented in the body diagram; the methods are the direct encoding of the corresponding methods specified in the body diagram.

The process classes are implemented as Java thread classes and the intermediate states, described by constructors in the behaviour graph, are implemented by the fields of the class; the unique method is run, whose code is determined by the behaviour graph of the class.

```
class DisplayDriver extends Thread{
  private String state = "Init"; // state constructor implementation
  private List_Digit the_dl; // digit list;
  private Display the_d; // the display;

  private Synch_Sign Show; // synchronous communication channels
  private Synch_Sign Take;
  private FileOutputStream Out;  // asynchronous communication channel

  DisplayDriver(Synch_Sign s, Synch_Sign t, Display d){
   super();
   Show = s;
     ... // initialization part continues
  }
  public void run(){
   while(true){
    Show.get(); // receives a signal
    Take.put(); // sends a signal
    the_dl= the_d.Read(); // reads the Display content
    state = "Dis"; // changes its state
     while (!the_dl.isEmpty()){
      Out.println(the_dl.Head());
      the_dl = the_dl.Tail();
     }}}
}
```

Fig. 8. Java implementation of DISPLAY_DRIVER

The communication channels of a process are fields referencing particular objects. A synchronous channel is implemented by using a special object that act as a "synchronizer". When a process P1 tries to synchronize with P2, it accesses the synchronizer to check whether P2 is ready for the synchronization. If P2 is not ready, then P1 is suspended. When P2 is ready, P1 is resumed and reads or writes the exchanged data. To ensure that only one process at a time gains the access to a channel method, as well as to suspend-resume processes we use the `synchronized` and the `wait-notify` mechanisms of Java.

The asynchronous channels are trivially implemented by Java streams. In the particular case of asynchronous communication among process, we use the specialized stream classes for pipeline communication. Moreover, if the communication among processes involves m writers and n readers, a particular object implements the connector in fig. 6(a) that continuously reads a data from anyone among the input channels and replicate it on each one of the output ones.

6 Conclusions and Related Works

We think that JTN could help to design complex systems using Java, even if in this paper we have used it on a really toy example, for the following reasons:

- it is strongly visual; we have tried to visually render the process behaviours, the system architecture, the way the components cooperate, as well as the definition of data operations and of object methods;
- the complexity and the intricacies of the systems is mastered by keeping separated data-types, objects and processes, and allowing to design such entities at the most abstract level compatible with a direct implementation in Java; for example, data-types are not objects and the user can define her data with the constructors of her choice to represent them. For example, if we want to concatenate two lists L1 and L2, we do not have to create two objects realizing L1 and L2 respectively, and then call the concatenation method on L1 (or L2); instead, we just apply the concatenation operation to terms representing L1 and L2 respectively.
- there is a direct correct encoding of the specification into a Java program that it is possible to make automatic by the use of some tool.

JTN is not purely OO, as it only includes some OO concepts, precisely those that are useful to model the features of the considered systems. We use classes and instances, plus inheritance and other relationships, to model the three kinds of constituents of the systems (data-types, passive and active components). The interactions among processes via shared memory is modelled by objects and method calls; encapsulation allows to control how processes access the objects in the shared memory.

Although JTN is Java-targeted, it is not useful only to produce Java code; indeed it can also be fruitfully used to model and design systems implemented by using another programming language, as ADA.

Note that JTN is not addressed to real-time systems, because the abstract specification method of [11] and the features of Java do not adequately support real-time programming.

It is possible to produce a full set of software tools to support the use of JTN: from interactive graphic editors to a static checker including the consistency check of the sequence diagrams with respect to the other diagrams, to browsers enhancing the hyper-textual aspects of the diagrams composing the specifications, to the translator into Java. We are considering also a form of debugger obtained by using a variant of the translation into Java. The execution of the modified program produces an output that can be transformed in a sequence diagram and so we can have a graphic presentation of the execution. The underlined required technology for the realization of such tools is easily available. At the moment we are looking for human resources to realize them.

Our future work will consider how to complete JTN; we want to investigate the structuring of processes and objects by introducing sub-components, a mechanism for the packaging of classes when one global class diagram is too large, other communication mechanisms, the notation for the description of the distribution level of the architecture and so on. The notion of inheritance for the process class requires further investigations too, with the determination of an associated type hierarchy.

To fully take advantage of JTN we need to propose a method for passing from the abstract specifications of [11] to the more detailed JTN ones, that is guidelines and hints that help the user to perform this task.

We are not aware of other "Java targeted" specification languages/notations for systems in the literature, even if there exit tools for generating Java code from generic object-oriented specifications (e.g., ROSE for UML).

To relate our proposal to other approaches we must first recall that JTN is not an OO specification language, but it is intended for reactive/concurrent/ distributed systems; this is the reason why it uses ingredients as processes strongly different from objects, system architecture and channels. However JTN encompasses a few OO concepts, for example, "object" as a way to encapsulate shared memory and "class" (for objects and processes) with inheritance as a way to modularly define "types" of objects and processes.

The JTN specifications are both graphic and formal, and in this respect JTN is similar to SDL [7] and Statecharts [5]; the differences with these two notations lay in the way the processes cooperate and in the paradigm followed for representing the process behaviour.

What said above shows also the differences/relationships with UML [9]: UML is OO, JTN is concurrency oriented; UML is a notation that can be used by many different development processes at different points, JTN is for Java targeted design of systems (companion formal/graphic notations for abstract design and requirement specifications have been developed, see [11, 10]); UML is semi-formal (precise syntax including well-formed conditions, semantics by English text), JTN is fully formal (it has a complete formal semantics because it can be easily transformed into a graphic-formal specification of [11], see [3]).

The use of data-types with constructors and of pattern matching in guarded commands come from ML [6], because we think that in many case that could be a compact and clear way to represent the data-types and their operations.

References

1. E. Astesiano and G. Reggio. Formally-Driven Friendly Specifications of Concurrent Systems: A Two-Rail Approach. Technical Report DISI–TR–94–20, DISI – Università di Genova, Italy, 1994. Presented at ICSE'17-Workshop on Formal Methods, Seattle April 1995.
2. E. Astesiano and G. Reggio. A Dynamic Specification of the RPC-Memory Problem. In *Formal System Specification: The RPC-Memory Specification Case Study*, number 1169 in LNCS. Springer Verlag, 1996.
3. E.Coscia and G.Reggio. JTN: the Reference Manual. Technical report, DISI – Università di Genova, Italy, 1999.
4. Gosling, Joy, and Steele. *The Java Language Specification*. Addison Wesley, 1996.
5. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8, 1987.
6. R. Harper, D. MacQueen, and R. Milner. Standard ML. Technical Report ECS-LFCS-86-2, LFCS-University of Edinburgh, 1986.
7. ITU. Z.100 ITU Specification and Description Language (SDL). Technical report, ITU, Geneva, 1993.
8. ITU. Z.120: Message Sequence Chart (MSC). Technical report, ITU, Geneva, 1993.
9. RATIONAL. UML Notation Guide Version 1.1. Available at `http://www.rational.com/uml/html/notation/`, 1997.
10. G. Reggio. A Method to Capture Formal Requirements: the INVOICE Case Study. In *Int. Workshop Comparing Specification Techniques*. Universite de Nantes, 1998.
11. G. Reggio and M. Larosa. A Graphic Notation for Formal Specifications of Dynamic Systems. In *Proc. FME 97*, number 1313 in LNCS. Springer Verlag, 1997.

Toward an Evolutionary Software Technology

Maritta Heisel
Fakultät für Informatik
Universität Magdeburg
D-39016 Magdeburg, Germany
Fax: (49)-391-67-12810
heisel@cs.uni-magdeburg.de

Extended Abstract

1 Motivation

Existing software engineering techniques usually treat the case where a *new* software system has to be built. All documents are developed from scratch, without any reference to existing documents. However, this situation is no longer realistic, because in more and more software projects, no new systems are constructed, but existing systems are evolved and adapted to new requirements. Hence, a task that becomes more and more important is to engineer *existing* software. Methods for an evolutionary software technology are still missing, even though object orientation and component-based software engineering enhance the possibility to re-use existing software.

This paper does not present finished results but explores some paths that lead to a software technology tailored for the evolution of existing systems.

2 Basic Principles

Every software product is made up of several documents, for example requirements documents, code, user manuals, etc. An important question for an evolutionary software technology is the choice of an appropriate basis for system evolution. Which documents should be the starting point of evolution strategies?

In the end, the evolution of a software system leads to changing its code. Therefore, one possible approach is to base software evolution strategies on the code. However, this approach is not advisable for the following reasons:

- The motivation for changing the existing system are additional or changed *requirements*. Hence, the requirements must be taken into account when evolving a system. A situation where the code is changed without any explicit reference to a requirements or specification document is unacceptable, even if it may common practice today.

- Before changing the system, the consequences of the change should be analyzed. It may be the case that new requirements interfere with old requirements. Such an analysis is highly non-trivial, even if it is performed on a high-level representation of the system. Trying to perform it on the code, which is the most low-level document representing the software system, would make the task even more difficult.

- The different documents that make up the software system, such as requirements, specification, and code, must be kept consistent. An evolution strategy that is based on code will almost certainly lead to neglecting the other documents. The result would be an undocumented and hence unmaintainable system.

We conclude that system evolution strategies should be based on *abstract* descriptions of software systems, i.e., requirements or specifications. Usually, these are informal documents expressed in natural language. To obtain semantically well-founded and automatable system evolution strategies, however, one should choose *formal* documents as a starting point. It follows that evolutionary software technology needs two phases:

1. Specification of existing systems
   This phase establishes the prerequisites for a systematic evolution of the system.

2. Systematic system evolution
   This phase deals with how to evolve a system in a systematic manner.

In the following sections, we sketch an approach how to tackle these two tasks.

3 Phase 1: Specification and Structuring of Existing Software

In his article "Software Aging" [Par93], Parnas describes how the structure of a software system is gradually destroyed by changes that are made when evolving or "maintaining" the system. For an evolutionary software technology, it is of utmost importance to preserve that structure when changes are made. This task is much easier when the structure, i.e., the software *architecture* [SG96], is made explicit.

We already mentioned that using the requirements and the code is indispensable for systematic software system evolution. With the architectural description, we have identified a third important document. This leads us to the idea to *construct different representations of the system and mappings between them.* On the one hand, we have the requirements of the system, which are its most abstract representation. On the other hand, there is the executable code, which is the most concrete representation of the system. In between the two, there are the specification and the architecture, as shown in Figure 1.

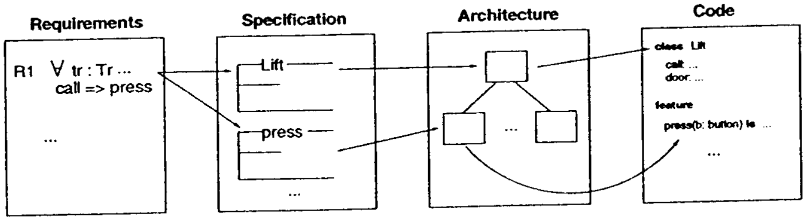


Figure 1: Representations and mappings

Having decided to use several different documents as the basis for system evolution, the tasks to be performed in the first preparatory phase of evolutionary software engineering consist in constructing different (formal) representations of the software system and mappings between these representations. The mappings, shown as arrows in Figure 1, constitute traceability links between the different parts of the various documents. For example, the mapping between the requirements and the specification shows for each requirement where it is reflected in the specification. Of course, the traceability links should be bi-directional. This means, for example, that it should not only be possible to find out how a requirement is distributed over the specification, but also to ask which requirements influenced the different parts of the specification.

Figure 1 just shows examples of possible intermediate representations of a system. One could also try to use fewer documents, for example do without the specification, or use more documents, for example the results yielded by reverse engineering tools as an additional representation between the architecture and the code [HK95]. The optimal number and nature of intermediate representations is still an open question. Too few intermediate representation result in very complex mappings, whereas too many documents result in an organizational overhead.

Another open question is *how* to construct the different representations and the mappings. A promising idea is to work from both ends, i.e., on the one hand from the requirements to the more concrete representations and on the other hand from the code to the more abstract representations. Both the requirements and the code should be available at the beginning of the first phase.

To validate the different representations and mappings, *consistency criteria* should be developed that help to detect errors early in the construction process.

If ever possible, the architectural description of the system should follow an *architectural style* [SG96]. Architectural styles characterize classes of systems that are structured according to the same principles. Architectural styles and concrete architectural descriptions, although usually represented as informal diagrams, can be formalized. In contrast to informal diagrams, formal architectural descriptions have a precise meaning. This makes it possible to define criteria for a concrete architecture to belong to an architectural style [HL97] and to define operations on architectures that accommodate the changes that are necessary during system evolution.

In practice, a legacy system will hardly be an instance of an architectural style, and it will also have other flaws. Hence, the first phase will not only consist in constructing additional documents, but it will lead to a first revision of the system in order to make it amenable to systematic evolution.

The first phase of evolutionary software engineering, as sketched in this section, should not depend on the languages that are used to express the various documents. Instead, our goal is to develop a methodology that is representation independent.

4 Phase 2: System Evolution

Once a software system is represented in the way sketched in Section 3, its evolution can be performed in a systematic way. First, the new requirements must be expressed. They can either replace old requirements or be additional requirements that enhance the functionality of the system.

Next, the consequences of adding or replacing requirements should be analyzed. The new requirements could be incompatible with the already existing requirements. Such a situation is called an *interaction*. This term was originally coined in telecommunications and referred to different *features* a customer can subscribe to. A feature interaction occurs when combining different features leads to undesired or unexpected behavior or logical contradiction. Hence, we call the analysis of the consequences of adding new requirements to an existing system *interaction analysis*.

If interactions between requirements are detected, they should be resolved before proceeding with the system evolution. Resolution can either be achieved by changing (usually weakening) the new requirements, or by revising existing requirements. This process is an iterative one and must be repeated until no more interactions are found.

Once the set of requirements has stabilized, i.e., all interactions are resolved, the new requirements must be incorporated in the system. If existing requirements are replaced by similar ones, we can follow the mappings constructed in the first phase and change the intermediate documents one by one. The mappings show the places where changes must be made. In this case, the overall structure of the system is not likely to change.

The situation is more complicated if the system functionality is enhanced by adding new requirements. Then, the mappings between the different documents no longer indicate the places where changes have to be made, and the present architecture of the system might no longer be appropriate. Methods are needed to

- exploit the mappings as far as possible also for new requirements.
  A possible approach is to classify the requirements, for example requirements that have to do with the user interface, or the update of data, etc. If a new requirement belongs to a class of requirements already present in the system, one could try to use the mappings belonging to that class.

- incorporate entirely new requirements into the system that have no similarity with existing requirements.
  It seems that this activity has something in common with the first phase of evolutionary software engineering. We must work from an updated set of requirements and incorporate new requirements into all of the following more concrete documents.

- change a software architecture in a systematic way.
  The change can either lead to a different architecture adhering to the same architectural style, or even entail a change of the architectural style of the system. To change software architectures, operators that work on architectural descriptions should be developed. These operators should also help to change the code.

So far, we have presented a general approach to evolutionary software engineering and have pointed out concrete research questions suggested by that approach. In the rest of the paper, we present a piece of work that is more mature than what was discussed before, and that makes up an important part of evolutionary software technology. That work is a heuristic algorithm to detect interactions in requirements. Such an algorithm is necessary for systematic software evolution, because the consequences of a system change must be analyzed before actually executing it.

4.1 Analyzing Requirements for Interactions

Given a set of already accepted requirements and a new requirement, the algorithm we present in the following calculates a set of candidate requirements with whom there might be an interaction. The algorithm is *heuristic*, which means that we cannot guarantee that all existing interactions are indeed detected. A heuristic algorithm is appropriate, because the notion of interaction can hardly be formalized. It covers more phenomena than just logical inconsistency[1]. Striving for a provably correct and complete algorithm would necessitate a formal and decidable notion of interaction. However, it is questionable if such a definition is possible or even desirable.

The algorithm determines a set of candidates to examine. It does not prove that there really is an interaction between the new constraint and each candidate. It is up to the stakeholders of the system to decide if the combination of the new requirement with the candidates yields an unwanted behavior or if it even is contradictory.

This algorithm was not developed specifically for software evolution, but as part of a requirements engineering method [HS99]. For analyzing interactions, however, it does not make any difference whether some of the requirements are already implemented or not. Hence, the algorithm is just as useful for the evolution of systems as it is for new systems.

System view.

We take the following view of a system: the system is started in some state S_1. When event e_1 happens at time t_1, then the system enters state S_2, and so forth:

$$S_1 \xrightarrow[t_1]{e_1} S_2 \xrightarrow[t_2]{e_2} \ldots S_n \xrightarrow[t_n]{e_n} S_{n+1} \cdots$$

An event can either come from the environment of the software system and be detected via sensors, or it can be the call of a system operation by a user. Hence, this view of a system is valid for both reactive and transformational systems.

Formalization of requirements.

Our requirements engineering method proposes to express requirements as constraints over the set *Tr* of admissible system traces, using *event* and *predicate* symbols.

We recommend to express – if possible – constraints as implications, where either the precondition of the implication refers to an earlier state or an earlier point in time than the postcondition, or both the pre- and postcondition refer to the same state, i.e. we have an invariant of the system.

Example. We consider an elevator. A possible requirement is: "When the lift passes by floor k, and there is a call from this floor, then the lift will stop at floor k".

$$\begin{array}{l} \forall\, tr : Tr;\ k : Floor \bullet \forall\, i : \mathrm{dom}\, tr \mid i \neq \#tr \bullet \\ \quad passes\_by(tr(i).s, k) \land call(tr(i).s, k) \Rightarrow tr(i+1).e = stop(k)) \end{array}$$

The symbols *passes_by* and *call* are predicate symbols, whereas the symbol *stop* is an event symbol. For each trace tr and each i-th element $tr(i)$ of the trace which is not the last one $(i \neq \#tr)$, we require that if the predicates *passes_by* and *call* are true of the state $tr(i).s$ and the floor k, then the next event $tr(i+1).e$ will be *stop*.

Schematic expressions.

The algorithm to determine interaction candidates uses schematic versions of formalized constraints. These schematic expressions have the following form:

$$x_1 \diamond x_2 \diamond \ldots \diamond x_n \leadsto y_1 \diamond y_2 \diamond \ldots \diamond y_k$$

where the x_i, y_j are literals (i.e., either predicate or event symbols or their negations) and each $\diamond$ denotes conjunction or disjunction. The symbol $\leadsto$ separates the precondition from the postcondition.

[1] Example: In the case study of an access control system [SH00b], we had the following requirements: "when the door is unblocked, it will be re-blocked after 30 seconds" and "when a person has entered the building, the door will be re-blocked". These requirements interact, because it is intended to block the door immediately after the person has entered and not only after 30 seconds. Logically, however, the two requirements are not contradictory. It would suffice to re-block the door after 30 seconds, no matter if the person has entered or not.

For transforming a constraint into its schematic form, we abstract from quantifiers and from parameters of predicate and event symbols. This results in a (deliberate) loss of information. For example, $x \wedge \neg x$ are no longer contradictory, because x could refer to a different argument than $\neg x$.

Note that a detailed formalization of a requirement is not strictly necessary to set up the schematic expressions. The schematic constraint could also be obtained directly from the natural-language requirement.

Example. The above requirement has the schematic form $passes\_by \wedge call \leadsto stop$

Semantic relations.

Because the set of interaction candidates is determined completely automatically, the algorithm cannot be based on syntax alone. We also must take into account the semantic relations between the different symbols. To that end, we construct three tables of semantic relations:

1. Necessary conditions for events. If an event e can only occur if predicate literal pl is true, then this table has an entry $pl \leftsquigarrow e$.

   Example. The event *stop* can only occur if the elevator is not halted: $\neg\, halted \leftsquigarrow stop$

2. Events establishing predicates. For each predicate literal pl, we need to know the events e that establish it: $e \leadsto pl$

   Example. The predicate *halted* is established by the event *stop*: $stop \leadsto halted$

3. Relations between predicate literals. For each predicate symbol p, we determine:

   - the set of predicate literals it entails: $p_{\Rightarrow} = \{q : PLit \mid p \Rightarrow q\}$
     Example. $halted_{\Rightarrow} = \{at, \neg\, passes\_by\}$
   - the set of predicate literals its negation entails: $\neg\, p_{\Rightarrow} = \{q : PLit \mid \neg\, p \Rightarrow q\}$
     Example. $\neg\, halted_{\Rightarrow} = \{passes\_by, \neg\, at\}$

Determining interaction candidates.

Two constraints are interaction candidates for one another if they have overlapping preconditions but incompatible postconditions, as is illustrated in Figure 2. "Incompatible" does not necessarily mean "logically inconsistent"; it could also mean "inadequate" for the purpose of the system.

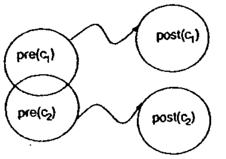


Figure 2: Interaction candidates

Our algorithm to determine interaction candidates consists of two parts: precondition interaction analysis determines constraints with preconditions that are neither exclusive nor independent of each other. This means, there are situations where both constraints might apply. Their postconditions have to be checked for incompatibility. Postcondition interaction analysis, on the other hand, determines as candidates the constraints with incompatible postconditions. If in such a case the preconditions do not exclude each other, an interaction occurs.

Precondition interaction candidates. If two constraints[2] $\underline{x} \leadsto \underline{y}$ and $\underline{u} \leadsto \underline{w}$ have common literals in their precondition ($\underline{x} \cap \underline{u} \neq \emptyset$), then they are certainly interaction candidates.

But the common precondition may also be hidden. For example, if $\underline{x}$ contains the event e, $\underline{u}$ contains the predicate literal pl, and e is only possible if pl holds ($pl \leftsquigarrow e$), then we also have detected a common precondition between the two events.

The common precondition may also be detected via reasoning on predicates. If, for example, $\underline{x}$ contains the predicate literal pl, $\underline{u}$ contains the predicate literal q, and there is a predicate literal w with $pl \Rightarrow w$ and $q \Rightarrow w$, then w is a common precondition.

[2] Underlined identifiers denote sets of literals.

Figure 3 shows how to calculate interaction candidates $C_{pre}(c', far)$ by a precondition analysis for a new constraint c' with respect to the set far of facts, assumptions, and requirements already defined.

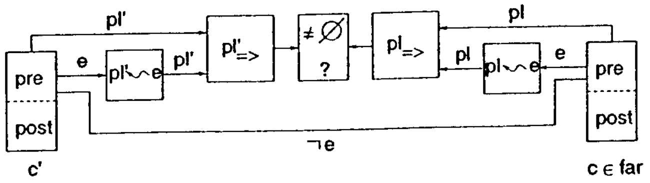


Figure 3: Determining interaction candidates by precondition analysis

Postcondition interaction candidates. To find conflicting postconditions, we compute the set of predicate literals that are entailed by the postcondition of the constraints under consideration. For an event e contained in the postcondition of a constraint, all predicate literals pl with $e \leadsto pl$ must be considered, too. If the postcondition of the new constraint entails a predicate literal whose negation is entailed by the postcondition of an already accepted constraint, these constraints are interaction candidates for each other. Figure 4 illustrates the definition.

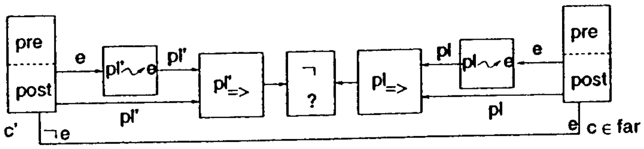


Figure 4: Determining interaction candidates by postcondition analysis

The algorithm is explained in more detail in [HS98b]. There, also the formal definitions of the candidate sets are given.

Example. To briefly illustrate the algorithm, we consider a new requirement: "The lift gives priority to calls from the executive landing", whose schematic form is

$$call \leadsto next\_stop\_at\_executive\_floor$$

This requirement interacts with the one previously given: if there is a call from the executive floor, the elevator will not necessarily stop at the floor it currently passes by, even if there is a call for that floor. The algorithm correctly identifies the previously given requirement as an interaction candidate for the new one via the common precondition *call*.

The algorithm has been validated in several case studies. Besides the elevator [HS98a], we have treated a microwave oven, an automatic teller machine, a simple telephone system, an access control system [SH00b] and a light control system [SH00a].

5 Conclusions

In this paper, we have presented a general approach to evolutionary software engineering. This approach must be refined, and the research questions raised need further investigation. In summary, we consider the following points as important:

- Mastering systematic software evolution becomes more and more important.
- Software evolutions strategies should not be based on code, but on more abstract representations of the software system.
- These abstract representations will usually not exist for legacy systems. Hence, the first task to perform before systematic software evolution is possible is to construct these representations.
- Several representations of a system on different levels of abstractions seem to be useful. Bi-directional traceability links must be established between these representations.

- Before a system is changed, it should be analyzed if the new requirements do not interact in an undesirable way with the rest of the system. We have presented a heuristic algorithm to support this task.
- Once the new requirements have stabilized, the system can be changed in a systematic way, making use of the mappings constructed in the first phase. This task is more difficult when the new requirements make a re-structuring of the system necessary.
- Re-structuring a system should be supported by architectural operators.
- In connection with the mapping from the architectural description to the code, the architectural operators should help in changing the code according to the architectural changes.

References

[HK95] Maritta Heisel and Balachander Krishnamurthy. Bi-directional approach to modeling architectures. Technical Report 95-31, Technical University of Berlin, 1995.

[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.

[HS98a] Maritta Heisel and Jeanine Souquières. Detecting feature interactions – a heuristic approach. In G. Saake and Can Türker, editors, *Proc. of the first FIREworks Workshop*, Preprint 10/98, pages 30–48, Fakultät für Informatik, 1998. Univ. Magdeburg.

[HS98b] Maritta Heisel and Jeanine Souquières. A heuristic approach to detect feature interactions in requirements. In K. Kimbler and W. Bouma, editors, *Proc. 5th Feature Interaction Workshop*, pages 165–171. IOS Press Amsterdam, 1998.

[HS99] Maritta Heisel and Jeanine Souquières. A method for requirements elicitation and formal specification. In Jacky Akoka, Mokrane Bouzeghoub, Isabelle Comyn-Wattiau, and Elisabeth Métais, editors, *Proceedings 18th International Conference on Conceptual Modeling, ER'99*, LNCS 1728, pages 309–324. Springer-Verlag, 1999.

[Par93] D. L. Parnas. Software Aging. In *Proceedings International Conference on Software Engineering*. ACM Press, 1993.

[SG96] Mary Shaw and David Garlan. *Software Architecture*. IEEE Computer Society Press, Los Alamitos, 1996.

[SH00a] Jeanine Souquières and Maritta Heisel. A method for systematic requirements elicitation: Application to the light control system. Technical Report A00-R-090, LORIA, Nancy, France, 2000.

[SH00b] Jeanine Souquières and Maritta Heisel. Une méthode pour l'élicitation des besoins: application au système de contrôle d'accès. In Yves Ledru, editor, *Proceedings Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL'2000*, pages 36-50. LSR-IMAG, Grenoble, 2000. http://www-lsr.imag.fr/afadl/Programme/ProgrammeAFADL2000.html.

Run-time Monitoring and Steering based on Formal Specifications*

Sampath Kannan, Moonjoo Kim, Insup Lee[†],
Oleg Sokolsky, and Mahesh Viswanathan
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA, U.S.A

August 30, 2000

Abstract

We describe the Monitoring-aided Checking and Steering (MaCS) framework that assures the correctness of software execution at run-time. Checking is performed based on a formal specification of system requirements to ensure that the current system behavior is in compliance with these requirements. When the system behavior violates these requirements, steering is invoked to correct the system. Our framework bridges the gap between formal verification and testing. The former is used to ensure the correctness of a design specification rather than an implementation, whereas the latter is used to validate an implementation. The paper presents an overview of the framework and the three scripting languages, which are used to specify what to observe from the running program, the requirements that the program should satisfy, and how to steer the running program to a safe state. An important aspect of the framework is clear separation between the implementation-dependent description of monitored objects and the high-level requirements specification. Another salient feature is automatic instrumentation of executable code for monitoring and steering. This paper also describes our current prototype implementation in Java.

1 Introduction

The design analysis and verification of distributed and real-time systems has become an important research topic over the past two decades. Important results have been achieved, in particular, in the area of formal verification [4]. Formal methods of system analysis allow developers to specify their systems using mathematical formalisms and prove properties of these specifications. These formal proofs increase confidence in correctness of the system's behavior. Complete formal verification, however, has not yet become a practical method of analysis. The reasons for this are twofold. First, the complete verification of real-life systems remains infeasible. The growth of software size and complexity seems to exceed advances in verification technology. Second, verification results apply not to system implementations, but to formal models of these systems. That is, even if a design has been formally verified, it still does not ensure the correctness of a particular implementation of the design. This is because an implementation often is much more detailed, and also may not strictly follow the design.

One way that people have traditionally tried to overcome this gap between design and implementation has been to test an implementation on a predetermined set of input sequences. This approach, however, fails to provide guarantees about the correctness of the implementation since not all possible behaviors can be tested. For mobile code, testing may not even be possible, especially if such code is downloaded on demand for execution. Conse-

*This research was supported in part by ARO DAAG55-98-1-0393, ARO DAAG55-98-1-0466, NSF CCR-9619910, NSF CCR-9988409, and ONR N00014-97-1-0505 (MURI).

[†]POC: lee@cis.upenn.edu

quently, when the system is running, it is hard to guarantee whether or not the system is executing correctly.

Computer systems are often monitored for performance measurement, evaluation and enhancement as well as to help debugging and testing [24]. Lately, there has been increasing attention from the research community to the problem of designing monitors that can be used to assure the correctness of a system at runtime [1, 5, 23, 19, 22, 17, 15]. These systems, however, tend to be based on informal specifications, require manual instrumentation, or depend much on the specificity of target systems. Our goal is to develop the monitoring, checking and steering framework based on formal specifications, which supports automatic instrumentation and isolates the implementation-dependency of the target system.

The overall structure of the Monitoring, Checking and Steering framework is shown in Figure 1. The user specifies the requirements of the system, which are expressed in terms of a sequence of abstract events, or trace. A *monitoring script* describes the mapping from observations to abstract events. The Monitor use this script to decide when and how to observe the system to extract abstract events needed by the checker. The Checker verifies the sequence of abstract events with respect to the requirements specification, detects violations of requirements and generate a meta-event as the result. The Steerer uses the sequence of meta-events to decide how to adjust the system dynamically to a safe state through control events.

In the next section, we decribe the framework. In keeping with the design philosophy of the framework, we have developed three languages in our prototype implementation. The Meta-Event Definition Language (MEDL) is used to express requirements. MEDL is based on an extension of a linear-time temporal logic. It allows us to express a large subset of safety properties of systems, including real-time properties. MEDL is described in Section 3 Monitoring scripts are expressed in the Primitive Event Definition Language (PEDL). PEDL describes primitive high-level events and conditions in terms of system objects. PEDL, therefore, is tied to the implementation language of the monitored system in the use of object names and types. MEDL is independent of the monitored system. The Steering Action Definition Language (SADL) is used to specify how a system is affected by steering actions. Section 4 describes the prototype implementation for Java as well as PEDL and SADL.

2 Overview of the Framework

The Monitoring, Checking and Steering (MaCS) framework specifies components that are necessary to perform run-time correctness monitoring of a system. It is independent of the system implementation. Of course, any concrete implementation of the framework will have to interface with the system to ensure proper exchange of information between the system and the monitor. In describing the framework, we carefully separate system-dependent components from system-independent ones. System-dependent components are presented in the context of an existing prototype implementation of the framework.

The overall structure of the MaCS framework is shown in Figure 1. The user specifies the requirements of the system in a formal language. Requirements are expressed in terms of high-level events and conditions (see Section 3). In addition, a *monitoring script* relates these events and conditions with low-level data manipulated by the system at run time. Based on the monitoring script, the system is *automatically* instrumented to deliver a stream of observations to the *monitor*. Observations are low-level data such as values of variables, method calls, etc. The monitor, also generated from the monitoring script, transforms this low-level data into abstract events. Since detection of abstract events is the primary function of the monitor, we also refer to it as the *event recognizer*.

The reason for keeping the monitoring script distinct from the requirements specification is to maintain a clean separation between the system itself, implemented in a certain way, and high-level system requirements, independent of a concrete implementation. Implementation-dependent event recognition performed by the monitor insulates the requirement checker from the low-level details of the system implementation. This separation also allows us to perform monitoring of heterogeneous distributed systems. A separate event recognizer may be supplied for each module in such system. Each event recognizer may process the low-level data in a different way, and all deliver high-level

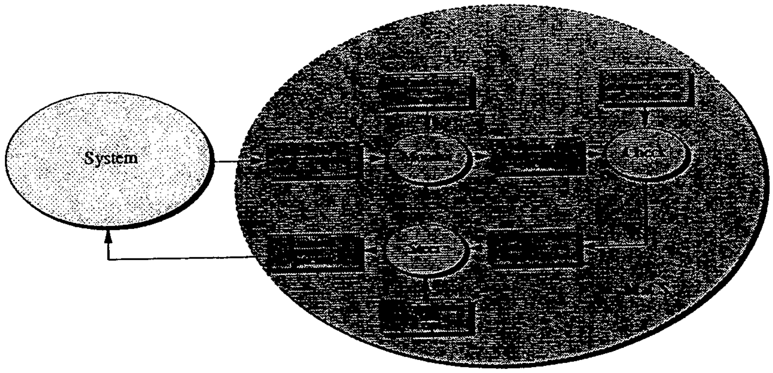


Figure 1: The Monitoring, Checking and Steering Framework

events to the checker in a uniform fashion.

The abstract events recognized by the monitor are delivered to the *run-time checker*. The run-time checker verifies the sequence of abstract events with respect to the requirements specification and detects violations of requirements. When a violation is detected, the checker raises an alarm. Besides the identification of the violation, the checker may be able to provide diagnostic information to the user, based on the data collected during monitoring. Because of the rich content, the outputs of the checker are called *meta-events*.

Run-time monitoring and checking effectively detects violations of system requirements and raise an alarm when a violation happens. The question is what to do when such a violation is detected, especially for those systems that cannot be reset and restarted. For such systems the run-time state can be adjusted to *steer* the system to a safe state through feedback from the checker to the monitored system. The design philosophy of steering follows the general idea of the framework, namely, that the system is mostly correct, except for a few subtle cases. Therefore, the steerer should not try to take over the control of the system, but help the system to recover from the detected violation by tuning parameters of the system. This approach captures the limitations of control that can be performed by a loosely coupled component such as the checker.

3 Logic for Events and Conditions

The framework provides an architecture for analyzing systems formally and flexibly using runtime information. In order to specify safety properties that are being ensured, we distinguish observations into events and conditions as in SCR [10]. *Events* occur instantaneously during the system execution, whereas *conditions* are predicates that hold for a duration of time. The distinction between events and conditions is very important in terms of what the monitor can infer about the execution based on the information it gets from the filter. The checker assumes that truth values of all conditions remains unchanged between updates from the monitor. For events, the checker makes the dual assumption, namely, that no events (of interest) happen between updates.

Since events occur instantaneously, we can assign to each event the time of its occurrence. Timestamps of events allow us to reason about timing properties of monitored systems. A condition, on the other hand, has *duration*, an interval of time when the condition is satisfied. There is a close connection between events and conditions: the start and end of a condition's interval are events, and the interval between any two events can be treated as a condition. This relationship is made precise in the logic [14].

Based on this distinction between events and conditions, we have a simple two-sorted logic. The syntax of conditions (C) and events (E) is as follows:

```
<C> ::=   c
        | [ <E>, <E> )
        | ! <C>
        | <C> && <C>
        | <C> || <C>
        | <C> => <C>
<G> ::= <E> -> <Statements>
<E> ::=   e
        | start(<C>)
        | end(<C>)
        | <E> && <E>
        | <E> || <E>
        | <E> when <C>
```

Here e refers to primitive events that are reported in the trace by the monitor; c is either a primitive condition reported in the trace or a boolean condition defined on the auxiliary variables. Guards (G) are used to update auxiliary variables that may record something about the history of the execution.

The models for this logic are similar to those for linear temporal logic, in that they are sequences of worlds. The worlds correspond to instants in time at which we have information about the truth values of primitive conditions and events. Each world is, therefore, labeled by the time instant it corresponds to and the set of primitive conditions and events that are true at that instant. Intuitively, these worlds correspond to the times when the monitor adds something to the trace.

The intuition in describing the semantics of events and conditions based on such models, is that conditions retain their truth values in the duration between two worlds, while events are present only at the instants corresponding to certain worlds. The labels on the worlds give the truth values of primitive conditions and events. The semantics for negation (!c), conjunction (c1 && c2), disjunction (c1 || c2) and implication (c1 => c2) of conditions is defined naturally; so !c is true when c is false, c1 && c2 is true only when both c1 and c2 are true, c1 || c2 is true when either c1 or c2 is true, and c1 => c2 is true if c2 is true whenever c1 is true. Conjunction (e1 && e2) and disjunction (e1 || e2) on events is defined similarly. Now, since conditions are true from some time until just before the instant when they become false, two events can naturally be associated with a condition, namely the instant when the condition becomes true (start(c)) and the instant when the condition becomes false (end(c)). Any pair of events define an interval of time, and forms a condition [e1, e2) that is true from event e1 until e2. The event e when c is true if e occurs and condition c is true at that time instant. Finally, a guard e -> stmt, is executed when event e is true; the effect of the execution is to update the values of the auxiliary variables according to the assignments given in stmt. The formal semantics for this logic is given in [13, 14].

Notice that some natural equivalences hold in this logic. For example, for any condition c, c = [start(c), end(c)). This allows one to identify conditions with pairs of events. Also, for conditions c1 and c2, and event e, e when c1 when c2 = e when c1 && c2.

3.1 Meta Event Definition Language (MEDL)

The safety requirements that need to be monitored are written in a language called MEDL. Like PEDL, MEDL is also based on the logic for events and conditions. Primitive events and conditions in MEDL scripts are imported from PEDL monitoring scripts; hence the language has the adjective "meta".

Auxiliary Variables. The logic described earlier has limited expressive power. For example, one cannot count the number of occurrences of an event, or talk about the *i*th occurrence of an event. For this purpose, MEDL allows the user to define auxiliary variables, whose values may then be used to define events and conditions. Auxiliary variables must be of one of the basic types in Java. Updates of auxiliary variables are triggered by events. For example,

RaisingGate -> {t = time (RaisingGate);}

records the time of occurrence of event RaisingGate in the auxiliary variable t. Expression

e1 -> {count_e1 = count_e1 + 1;}

counts occurrences of event e1. A special auxiliary variable currentTime can be used to refer to the current time of the system. It is set to be the timestamp of the last message received from the filter.

Defining events and conditions. The primitive events and conditions in MEDL are those that are defined in PEDL. Besides these, primitive conditions can also be defined by boolean expressions using the auxiliary variables. More complex events and conditions are then built up using the various connectives described in Section 3. These events and conditions are then used to define safety properties and alarms.

Safety Properties and Alarms. The correctness of the system is described in terms safety properties and alarms. Safety properties are conditions that must *always* be true during the execution. Alarms, on the other hand, are events that must never be raised. Note that all safety properties [16] can be described in this way. Also observe that alarms and safety properties are complementary ways of expressing the same thing. The reason we have both of them is because some properties are easier to think of in terms of conditions, while others are easier to think of in terms of alarms.

The checker, which is generated automatically from the MEDL script, evaluates the events and conditions described in the script, whenever it reads an element from the trace. The evaluation of individual events and conditions is fairly standard based on the semantics of the logic. However, there are dependencies between different events and conditions. For example, an event e1 that is defined in terms of an auxiliary variable that is updated by event e2, must be evaluated after e2 and the variable have been updated. Hence, the checker must evaluate the events and conditions in a consistent order. In our implementation we use a DAG data structure that implicitly encodes this dependency and has additional information that allows for fast evaluation of the events and conditions. Details of this algorithm can be found in [13].

Example. We illustrate the use of MEDL using a simple but representative example. The example is inspired by the railroad crossing problem, which is routinely used as an illustration of real-time formalisms [9]. The system is composed of a gate that can open and close, taking some time to do it, trains that pass through the crossing, and a controller that is responsible for closing the gate when a train approaches the crossing and opening it after it passes. The common specification approach is to assume an upper bound on the time necessary for the gate to open or close. In reality, however, mechanical malfunctions may result in unexpectedly slow operation of the gate. A timely detection of such a violation lets the train engineer stop the train before it reaches the crossing. In this example, we monitor the controller of the gate, using the requirement that the gate is down within 30 seconds after signal *CloseGate* is sent, unless signal *OpenGate* is sent before the time elapses. Precisely, we check that if there is a signal *CloseGate*, not followed by either signal *OpenGate* or completion of gate closing, is present in the execution trace, then the time elapsed since that signal is less than 30.

```
import event OpenGate, CloseGate;
import condition Gate_Down;
//Declaration of auxilliary variables
    var float lastClose;
    var float currentTime;
//Safety properties
    property GateClosing =
        [ CloseGate when !Gate_Down,
          OpenGate || start(Gate_Down) )
        => lastClose + 30 > currentTime;
//Rules for updating auxilliary variables
    CloseGate ->
        {lastClose = time(CloseGate); }
```

Figure 2: A sample MEDL script

The correctness requirement for the gate is given in the MEDL script shown in Figure 2. The time of the last occurrence of event `CloseGate` is recorded by the auxiliary variable `lastClose`. The requirement uses the events and conditions imported from the monitoring script and states that if there was a `CloseGate` event at the time when the gate was not down, which was not followed by either event `OpenGate` or condition `Gate_Down` becoming true, then the time allotted for gate closing has not elapsed yet.

4 Java MaCS

A prototype of the framework has been implemented and tested on a number of examples. The prototype is targeted towards monitoring and checking of programs implemented in Java. Java has been chosen as the target implementation language because of the rich symbolic information that is contained in Java class files, the executable format of Java programs. This information allows us to perform the required instrumentation easily and concentrate on the more fundamental aspects of the monitoring and checking framework implementation. Figure 3 shows the overall structure of the Java-based MaCS prototype.

The PEDL language of the prototype allows the user to define primitive events in terms of the objects of a Java program: updates of program variables (fields of a class or local variables of a method) and method calls. Automatic instrumentation guarantees that all relevant updates are detected and propagated to the event recognizer.

The prototype uses interpreters for PEDL and MEDL. Each interpreter includes a parser for the respective language and works on a parsed version (the abstract syntax tree) of a script. The MEDL interpreter is the run-time checker. It accepts primitive events sent by the event recognizer and, after each primitive event, re-evaluates all events and conditions described in the MEDL script that may be affected by this event and raises alarms if necessary. If a steering action is invoked in response to an alarm, the run-time checker sends the corresponding message to the system. The PEDL interpreter is the event recognizer. It accepts the low-level data sent by the instrumented program and, based on the definitions in the monitoring script, detects occurrence of the primitive events and delivers them to the run-time checker. In addition, the PEDL interpreter produces the instrumentation data that is used to automatically instrument the system.

The MaCS instrumentation is based on JTREK class library [12], which provides facilities to explore a Java class file and insert pieces of bytecode, preserving integrity of the class. During instrumentation, the instrumentor detects updates to monitored variables and calls to monitored methods and inserts code to send a message to the event recognizer. The message contains the name of the called method and its parameter values, or the name of the updated variable and its new value. Each message contains a time stamp that can be used in checking of real-time properties. In addition, if steering is to be performed, the instrumentor inserts the additional code at the positions prescribed by the steering conditions. The code tests the flag for action invocations and makes calls to the injector to execute the action.

The parser for SADL produces two components: 1) a list of actions together with their conditions in the form that can be used by the instrumentor; 2) a new class, `Injector`, which is responsible for communication with the run-time checker. When the system is started, the injector is loaded into the virtual machine of the monitored system. At run time, when a steering action happens, the injector receives a message from the checker and sets a flag to indicate that the steering action has happened. The bodies of the steering actions are also represented in the prototype as methods of the `Injector` class.

During system start-up, the interpreters for PEDL and MEDL are run together with the system, either on the same computer or elsewhere on the network. Connections between the system and the interpreters are established during the system initialization.

We give a brief overview of the three languages, PEDL, MEDL, and SADL, used to describe what to observe in the program, the requirements the program must satisfy, and how to steer the running program, respectively. These languages are based on the logic for *events* and *conditions* described in Section 3.

4.1 Primitive Event Definition Language (PEDL)

PEDL is the language for writing monitoring scripts. The design of PEDL is based on the following two principles. First, we encapsulate all implementation-specific details of the monitoring process in PEDL scripts. Second, we want the process of event recognition to be as simple as possible. Therefore, we limit the constructs of PEDL to allow one to reason only about the current state in the execution trace. The name, PEDL, reflects the fact that the main purpose of PEDL scripts is to define primitive events of requirement specifications.

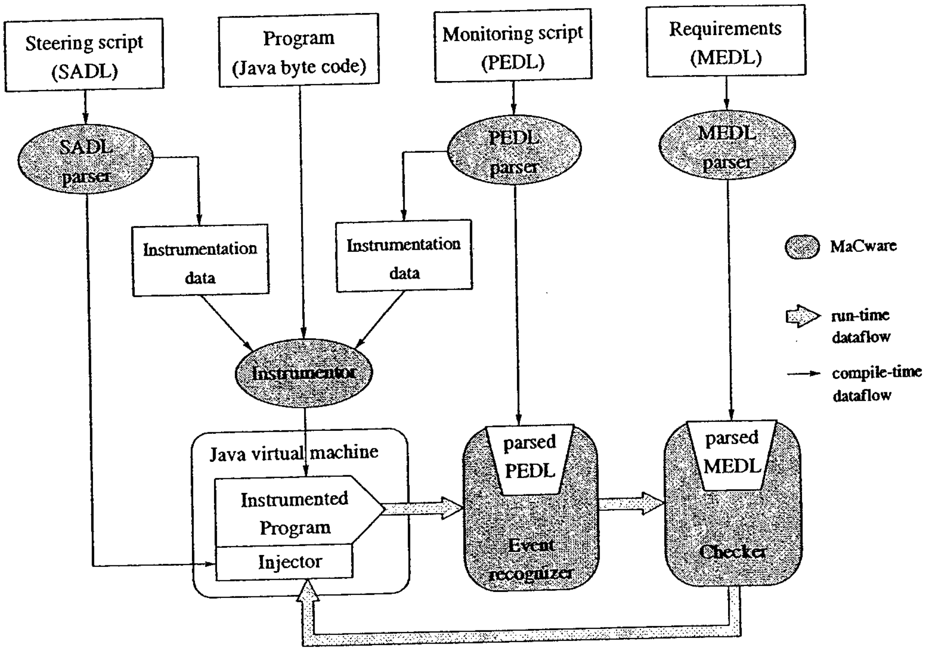


Figure 3: Java-based MaCS prototype

Monitored Entities. PEDL scripts can refer to any object of the target system. This means that declarations of monitored entities are by necessity specific to the implementation language of the system. In the current prototype which is based on Java, values of fields of an object, as well as of local variables of a method, and method calls can be monitored. Examples of monitored entities' declarations are given in Figure 5.

Defining Conditions. Primitive conditions in PEDL, are constructed from boolean-valued expressions over the monitored variables. An example of such condition is

```
condition TooFast =
      Train.Position().speed > 100.
```

In addition to these, we have primitive condition `InM(f)`. This condition is true as long as the execution is currently within method `f`. Complex conditions are built from primitive conditions using boolean connectives.

Defining Events. The primitive events in PEDL correspond to updates of monitored variables, and calls and returns of monitored methods. Each event has an associated timestamp and may have a tuple of values.

The event `update(x)` is triggered when variable x is assigned a value. The value associated with this event is the new value of `x`. Events `StartM(f)` and `EndM(f)` are triggered when control enters to and return from method `f`, respectively. The value associated with `StartM` is a tuple containing the values of all arguments. The value of an event `EndM` is a tuple that has the return value of the method, along with the values of all the formal parameters at the time control returns from the method. Besides these three, we have one other primitive event which is `IoM(f)`. This is also triggered when control returns from a method `f`, but has as its value a tuple that contains the return value of the method, and the values of the arguments *at the time of method invocation*. This event allows one to look at the input-output behavior of a method, and is needed if one wants to *program check* some numerical computation. Notice that event `IoM(f)` is the only event to violate our second design principle, namely that the operation of the event recognizer is to be based only on *the current state*.

```
class GateController {
    public static final int GATE_UP    = 0;
    public static final int GATE_DOWN  = 1;
    public static final int IN_TRANSIT = 2;
    int gatePosition;
    public void open()  { ... }
    public void close() { ... }
    ...
};
```

Figure 4: Implementation of the gate controller

```
  export event OpenGate, CloseGate;
  export condition Gate_Down;
//Monitored Methods
  monmeth void GateController.open();
  monmeth void GateController.close();
//Monitored variables
  monent int  GateController.gatePosition;
//Definition of conditions
  condition Gate_Down =
       (GateController.gatePosition ==
        GateController.GATE_DOWN);
//Definition of Events
  event OpenGate =
       StartM(GateController.open());
  event CloseGate =
       StartM(GateController.close());
```

Figure 5: A sample PEDL script

All the operations on events defined in the logic can be used to construct more complex events from these primitive events. In PEDL, we also have two attributes `time` and `value`, defined for events. As mentioned in Section 3, events have associated with them attribute values, and the time of their occurrence, and these can be accessed using the attributes `time` and `value`. `time(e)` gives the time of the last occurrence of event `e`, while `value(e)` gives the value associated with `e`, provided `e` occurs. `time(e)` refers to the time on the clock of the monitored system (which may be different from the clock of the monitor) when this event occurs.

Example. Continuing with the railroad crossing example, we illustrate the use of PEDL. Figure 4 shows a fragment of the gate controller implemented as a Java class. The state of the gate is represented as variable `gatePosition`, which can assume constant values `GATE_UP`, `GATE_DOWN`, or `IN_TRANSIT`. The controller controls the gate by means of methods `open()` and `close()`. For simplicity, we assume that there is only one instance of class `GateController` in the system.

We need to observe calls to methods `open()` and `close()`, and the state of the gate. The following PEDL script introduces high-level events `OpenGate`, `CloseGate` and condition `Gate_Down`.

4.2 A Language for Steering Actions (SADL)

Steering scripts let the user specify steering actions and control the moment when a steering action is executed, ensuring its effectiveness. This is done by means of steering conditions associated with each action. After a steering exception is raised by the checker, its execution is delayed until its condition is satisfied. Steering conditions can be either static or dynamic. Static conditions are fully evaluated during instrumentation, while dynamic conditions depend on run-time information. Dynamic conditions provide for finer control of action invocations. On the other hand, additional effort to evaluate the conditions at run time can adversely affect performance of the system. In the current prototype, only static conditions are implemented.

To specify steering actions, we designed a special scripting language SADL (Steering Action Definition Language). The steering scripts written in SADL specify how the system objects are affected by a steering action. Figure 6 shows a sample script, taken from a study in steering of artificial physics algorithms [8]. In the example, a pattern of particles is being formed by applying forces of attraction and repulsion between the particles. If a problem is discovered, the checker steers the system by manipulating the force of repulsion.

The script consists of two main sections: declaration of steered objects (that is, system objects that are involved in steering) and definition of steering actions where the declared objects are used. Since steering is performed directly on the system objects, SADL scripts are by necessity dependent on the implementation language of the target system. Since the MaCS prototype implementation aims at systems implemented in Java, SADL scripts used in the prototype are also tied to Java.

```
steering script mav

steered objects
   Air    MAV:air;
   Point  MAV:position;

steering action controlRepulsion( boolean tf )
   = { call (MAV:air).setRepulse(tf); }
     before write MAV:position;

end
```

Figure 6: A sample steering script

Steering entities. The entities involved in steering can be fields and methods of Java classes as well as local variables of methods. In the example, the steered entity is the `Air` object, a repository of the algorithm parameters shared by all particles. In addition, the variable representing position of a particle is used in the specification of the steering action.

Defining steering actions. The second section of the steering script defines steering actions and specifies steering conditions. An action can have a set of parameters that are computed by the checker and passed to the system together with the action invocation. The body of an action is a collection of statements, each of which is either a call to a method of the system or an assignment to a system variable. In the example, the steering action calls a method that controls repulsion between particles, and is allowed to happen every time the position of a particle is about to be updated.

```
ReqSpec mav

import action controlRepulsion(boolean);

alarm noPattern = ...;

noPattern -> { invoke controlRepulsion(true); }

end
```

Figure 7: Action invocation in the MEDL script

Invocation of steering actions in MEDL scripts. In addition to a steering script, the requirement specification language is extended to provide for invocation of steering actions. An action is invoked in response to an occurrence of an event or an alarm. Figure 7 presents a fragment of the MEDL script of the artificial physics example. It shows the declaration of the steering action `controlRepulsion`, imported from the steering script, and the alarm `noPattern` that is raised by the checker when it detects a violation of the pattern formation. The definition of the alarm is rather complex and is omitted for clarity. When the alarm is raised, the steering action is invoked with the `true` value of its parameter, which suspends repulsion between particles and triggers the process of restoring the pattern.

5 Related Work

The "behavioral abstraction" approach to monitoring was pioneered by Bates and Wileden [1]. Although their approach lacked formal foundation, it provided an impetus for future developments. Several other approaches pursue goals that are similar to ours. The work of [5] addresses monitoring of a distributed bus-based system, based on a Petri Net specification. Since only the bus activity is monitored, there is no need for instrumentation of the system. [20] generates a monitor for real-time reactive system based on a tabular requirement specification. The monitor watches over a pair of input and output of the system. The authors of [23] also consider only input/output behavior of the system. In our opinion, the instrumentation of key points in the system allows us to detect violations faster and more reliably, without sacrificing too much performance. The test automation approach of [19] is also targeted towards monitoring of black-box systems without resorting to instrumentation. In contrast, we aim at using the MaCS framework beyond testing, during real system executions. Sankar and Mandel have developed a methodology to continuously monitor an executing Ada program for specification consistency [22]. The user manually annotates an Ada program with constructs from ANNA, a formal specification language. Mok and Liu [17] proposed an approach for monitoring the violation of timing constraints written in the specification language based on Real-time Logic as early as possible with low-overhead. The framework we describe in this paper does not limit itself to any particular kind of monitored properties. In [15], an elaborate language for specification of monitored events based on relational algebra is proposed. Instrumentation of high-level source code is provided automatically. Collected data are stored in a database. Since the instrumentation code performs database queries, instrumentation can significantly alter the performance of a program.

A large body of related research work concentrates on automated generation of *test oracles* from the requirements. A general methodology for doing this is discussed in [21], together with examples in Real Time Interval Logic (RTIL) and Z. In [2] a trace analysis tool for LOTOS requirements is described, while [7] describes a similar tool for Estelle requirements. Generating test oracles for Graphical Interval Logic (GIL) is discussed in [6, 18]. An equivalent problem for a safe fragment of Linear Temporal Logic is put forth in [11]. This fragment is expressively similar to MEDL. We note that the MaCS framework gives more than just a test oracle for a given specification. Its ability to generate diagnostic information and provide feedback the the system in case of requirement violations makes it a more general tool.

The simulation and monitoring platform MT-Sim [3], based on the graphical real-time specification language Modechart, is similar in its intent to MaCS, however, we are not as tied to a fixed specification formalism.

6 Conclusions

This paper describes the Monitoring-aided Checking and Steering (MaCS) framework which is developed to assure the correctness of execution at run-time and to perform dynamic correction of system behavior by steering actions. Monitoring and checking is performed based on a formal specification of system requirements, and is use to detect violations of safety properties in the observed execution of the monitored system. Steering is integrated with monitoring and checking to put the system back to a safe state. The MaCS framework is a step towards bridging the gap between verification of system design specifications and validation of system implementations in a high-level programming language. The former is desirable but yet impractical for large systems, while the latter is necessary but informal or incomplete.

There are several issues that need further work. For example, we would like to understand better the theoretical basis for steering; in particular, what problems can be resolved by means of steering, what is the right way to reason about steering, to convince ourselves that steering will have the desired effect. The current prototype system for Java is available at `www.cis.upenn.edu/~rtg/macs`, and will serve as an important vehicle in exploring the possibilities and shortcomings of our approach. We are currently conducting several case studies in monitoring and steering of systems, and we expect to gain much experience from them. We also plan to extend the prototype implementation to support distributed systems written in Java.

References

[1] P.C. Bates and J.C. Wileden. High-level debugging: The behavioral abstraction approach. *J. Syst. Software*, 3(255-264), 1983.

[2] G.v. Bochmann, R. Dssouli, and J.R. Zhao. Trace analysis for conformance and arbitration testing. *IEEE Transactions on Software Engineering*, 15(11):1347–1356, November 1989.

[3] Monica Brockmeyer, Farnam Jahanian, Constance Heitmeyer, and Bruce Labaw. A flexible, extensible environment for testing real-time specifications. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*, 1997.

[4] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[5] Michel Diaz, Guy Juanole, and Jean-Pierre Courtiat. Observer - a concept for formal on-line validation of distributed systems. *IEEE Transactions on Software Engineering*, 20(12):900–913, December 1994.

[6] Laura K. Dillon and Q. Yu. Oracles for checking temporal properties of concurrent systems. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT'94)*, volume 19, pages 140–153, December 1994. Proceedings published as Software Engineering Notes.

[7] S.A. Ezust and G.v. Bochmann. An automatic trace analysis tool generator for estelle specifications. *Computer Communication Review*, 25(4):175–184, October 1995. Proceedings of ACM SIGCOMM 95 Conference.

[8] Diana Gordon, William Spears, Oleg Sokolsky, and Insup Lee. Distributed spatial control and global monitoring of mobile agents. In *Proceedings of the IEEE International Conference on Information, Intelligence, and Systems - ICIIS'99, to appear*, November 1999.

[9] C. Heitmeyer and D. Mandrioli, *Eds. Formal Methods for Real-Time Systems.* Number 5 in Trends in Software. John Wiley & Sons, 1996.

[10] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. Scr*: A toolset for specifying and analyzing requirements. In *Proceedings of COMPASS*, 1995.

[11] L. J. Jagadeesan, A. Porter, C. Puchol, J. C. Ramming, and L.G.Votta. Specification-based testing of reactive software: Tools and experiments. In *Proceedings of the International Conference on Software Engineering*, May 1997.

[12] Java Technology Center, Compaq Corp. *Compaq JTrek* . Online documentation: http://www.digital.com/java/download/jtrek/.

[13] Moonjoo Kim, Mahesh Viswanathan, Hanêne Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. A framework for runtime correctness assurance of real-time systems. Technical Report MS-CIS-98-37, University of Pennsylvania, 1998.

[14] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications*, 1999.

[15] Yingsha Liao and Donald Cohen. A specificational approach to high level program monitoring and measuring. *IEEE Transactions on Software Engineering*, 18(11):969–979, November 1992.

[16] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.

[17] Aloysius K. Mok and Guangtian Liu. Efficient run-time monitoring of timing constraints. In *IEEE Real-Time Technology and Applications Symposium*, June 1997.

[18] T.O. O'Malley, D.J. Richardson, and L.K. Dillon. Efficient specification-based test oracles. In *Second California Software Symposium (CSS'96)*, April 1996.

[19] J. Peleska. Test automation for safety-critical systems: Industrial application and future developments. In *FME'96: Third International Symposium of Formal Methods Europe*, volume 1051 of *LNCS*, pages 39–59, 1996.

[20] D. K. Peters and D. L. Parnas. Requirements-based monitors for real-time systems. In *ISSTA'00: International Symposium on Software Testing and Analysis*, 2000.

[21] D.J. Richardson, S. Leif Aha, and T.O. O'Malley. Specification-based oracles for reactive systems. In *14th International Conference on Software Engineering*, May 1992.

[22] Sriram Sankar and Manas Mandal. Concurrent runtime monitoring of formally specified programs. In *IEEE Computer*, pages 32 –41, March 1993.

[23] T. Savor and R. E. Seviora. An approach to automatic detection of software failures in real-time systems. In *IEEE Real-Time Technology and Applications Symposium*, pages 136 –146, June 1997.

[24] Beth A. Schroeder. On-line monitoring: A tutorial. In *IEEE Computer*, pages 72 – 78, June 1995.

Towards Practical Support for Component-Based Software Development Using Formal Specification

– – Position Statement – –

Heinrich Hussmann

Dresden University of Technology
Department of Computer Science
Email: h.hussmann@computer.org

Abstract

Starting from an analysis of the situation of a software developer using pre-fabricated components, it is investigated in which form techniques and formalisms from the area of formal specification can provide practical aid in development. Several different dimensions of precise component specification are identified. While formal specifications can be helpful for several of these dimensions, it is argued that the most relevant application area may be a flexible mechanism for creating different but consistent views on a complex system. Ideas for concrete tool support based on the Object Constraint Language (OCL) are sketched.

1 Introduction

Building software from pre-fabricated components has been discussed since a long time (e.g. [McIlroy68, Cox90] as the ultimate breakthrough towards industrial production of software. It took until the end of the 90s that a number of technologies have appeared, mainly based on object-oriented principles, which make the component-based construction of complex software practically viable. A good overview of the current state of the art is given in [Szyperski97]. The term "component" is used in several variants. For this paper, we are concentrating on those approaches to component-based software development, where (relatively small) components are designed and produced with the explicit and sole purpose of later composition to applications. Examples of important recent technologies of this kind are Java Beans [JavaBeans97], Delphi Components, Enterprise Java Beans [Monson-Haefel99], or CORBA components [CC99].

The normal working situation of an application developer in component-based programming is to place a number of components on a graphical design surface, to configure them by setting property values and to interconnect them by event mechanisms. Designing an application becomes very similar to drawing a diagram with a special software tool. Of course, the range of possible applications that can be produced in this way is limited by the possibilities for component configuration, i.e. the properties (parameters) of the pre-fabricated components and the foreseen support for creating events and reacting to events in a component.

It is relatively easy to deal with software components in this style as long as these components mirror concrete elements of the user interface (so-called *visual components*). So for the realisation of Graphical User Interfaces (GUIs), component-based approaches are already accepted as a standard technology and are supported by many commercial software development tools (e.g. JBuilder, Delphi, NetBeans). As soon as non-visual components are used, developers often feel less at ease with the component-based approach. Components acting as an information source for visual components (e.g. access to a database) are still relatively easy to integrate (and frequently used in practice). For most other kinds of components, in particular so-called business components representing facts and procedures of real-life businesses, significant effort has to be spent on learning the concepts that are behind the given components. However, the technology is mature enough to cover also server-side components, including issues of persistence, transaction and security (e.g. in Enterprise Java Beans).

As a starting point for the further discussion, let us analyse the situation of a software developer using the above-mentioned new technologies. The following observations can be made:

- When a specific use case is to be realised, the developer has to find the appropriate components to achieve the desired effect. In a large and sophisticated component library, the effort for finding an appropriate component and learning its use is quite high, often comparable to the effort to be spent for a direct implementation of the functionality. Search engines for component libraries can provide some help here. However, tools can never really save the effort that has to be spent on learning the actual application domain of the components and the conceptual ideas behind the components being on offer. Moreover, once a potentially suitable component has been found, it is extremely important to get precise information on the preconditions under which one can assume the component or a specific method of a component to work properly.
- The resulting network of interconnected components is not easy to understand. There is no obvious overall structure of the application "program". The design is created interactively, and is often produced in a series of iterations. The graphical tools are not sufficient for showing the logical structure: Showing all interconnections graphically tends to give the optical impression of a bowl of spaghetti. Not showing the interconnections makes it difficult to trace the dependencies among components.
- One of the main motivations for using components is an economic one. Based on component technology, the evolution of a market is expected where third-party software developers offer components to application developers. One of the crucial issues in this model is the contractual situation. A component somehow provides a contract stating that it delivers some functionality when the environment meets certain preconditions. But how can a software developer trust the quality of the software components he is buying and using, when these contracts are not made explicit?

As can be seen from the list above, the main problems in dealing with components lie in the adequate specification of components, in particular regarding their interaction. In the remainder of this text, this topic is discussed further. In section 2, it is claimed that formal specification techniques are a relevant technology for improving component technology. Section 3 contains seven theses on the way how some of the well-developed techniques from formal specification may be applied to bring practical aid for component-based software development. Section 4 sketches potential tool support following the ideas from section 3.

2 Formal Specification and Components

Extensive research in formal specification languages (i.e. specification languages providing a formal syntax with a mathematical semantics) has been carried out now for several decades. Interestingly, the effect of this research on practical software development has been low up to now. [Meyer97] argues that the main reason is that the effort spent on a precise specification does not pay back appropriately in the current market situation where software is produced with very high time pressure, relatively low quality standards and a very limited extent of reuse. He claims that the situation will change completely when component-based development is applied on a large scale.

- For component developers and component users, an urgent need exists to rely on precise contracts about the functionality of the software. To protect the component developers, software components are usually delivered as compiled code, so the source code cannot be used for clarifying issued about the detailed functionality. It is a clear strength of formal specification techniques to describe software as a "black box", but with absolute precision. So it seems logical to have a formal specification of a component (system) as part of the contract.
- For component developers, high investments into quality of the components pay back, since they can expect high-quality components to sell in very large quantities. So it may be economically feasible to spend extra effort on formal specification and even verification.

These ideas have led to the formation of an informal group of persons interested in high-quality components, the "Trusted Components Initiative" [TCI]. However, compared to the fast development of software technology (in particular in object-oriented and component-based approaches), it is astonishing that only relatively little progress has been made in this area during the last three years.

Further attempts to establish a precise component specification language have been made also by people from a less theoretical background, for instance the "BOCA" (Business Object Component Architecture Initiative) [Digre98]. BOCA aimed at defining a standardised component definition language where the semantics of components can be precisely specified by making reference to an underlying semantic model of the problem. This initiative somehow was torn apart by competing forces in the OMG standardisation process (the BOCA approach was rejected by OMG in summer 1998).

So altogether one can state that precise specifications for components have not yet taken up a role as a driving technology, despite of the potentially good perspective. One reason for the relatively slow progress may lie in the "fastly moving scenario" which is referred to in the motto of this workshop. The main reason for using components is to shorten development time and to ease application adaptation, but without introducing new (potentially time-consuming) technologies for quality assurance.

So it seems that simply specifying the behaviour of components with existing formal specification approaches is too naive an approach for being applied in practice. Below, we try to discuss more sophisticated and restricted ways how formal specification can be of practical help, keeping the "fastly moving scenario" in mind.

3 Seven Theses on the Practical Usefulness of Formal Specification for Components

In the following, a number of theses are put together regarding the potential of formal specification techniques for component specification.

3.1 Thesis 1: Components cannot be considered in isolation.

At a first look, it may seem logical to consider a component as an entity that is completely described by its (formally specified) interface. However, components tend to be tied together in specific groups quite closely. As an example from GUI components (like Java Swing), MenuBars are closely tied to MenuItems, Separators and other components. In business objects, an order processing component has to know about products and customers, which are most probably dealt with in other components. So when talking about precise specification of components, one is always talking about joint specification of a group of components (a *component-based application framework*).

The idea that was used in the BOCA approach seems to be logical here: A specification of a component system consists of a specification of the semantic domain plus individual contracts for components using the terms of the semantic domain.

3.2 Thesis 2: Full formal specification of business domain semantics is too much.

Based on the idea of a semantic domain specification, it seems to be necessary to fully specify the application domain of a group of specified components. However, this turns out as an extremely complex task, which is not needed in all circumstances. So for instance, specifying the exact behaviour of GUI components requires extensive descriptions of the geometry of windows. However, many of the operations are just straightforward in their semantics, like an action being called when a button is pushed. There is no doubt about the required functionality here, so formal specification of these aspects would not enhance the quality of component description. A similar situation arises in business objects, where many rules and algorithms, e.g. for accounting, are defined as standards of the business domain, so the only contract for the component needed is to refer to these (mostly informal) standards.

When considering the evolution of reuse in programs for a specific business domain, there an increase in abstraction levels can be observed (see e.g. [Pree 97]). In the beginning, solutions for individual problems of the application domain are implemented (and specified). In a later stage, libraries for frequently needed functions are designed, leading e.g. to object-oriented application frameworks. When such a framework is mature enough, it is possible to design a group of components for this application domain in such a ways that its adaptation to individual problems does not require any programming language code anymore. This means that the components are introduced at a level where the application domain is very well understood and where the components themselves mirror very closely the concepts of the application domain. So a relatively small semantic gap remains between the business domain and the component framework. The need for a formal specification on this level disappears mostly, since the concepts are thoroughly understood by the contract partners. There is no need for instance in components targeted at accounting systems to formally specify algorithms for computation with interest rates, since the relevant algorithms are well known to everybody working in this area and can just be referenced by using the appropriate term used in the business domain.

As explained above, there are two strong forces that prohibit the *full* formal specification of a business domain: the sheer effort required to do so, and the limited usefulness in a well-known business domain. However, *some aspects* of a system, in particular co-ordination aspects, require formal specification, see below. For practical application, it seems in most cases sufficient to give a semi-formal specification of the full business domain. This means to define e.g. UML class diagrams plus a number of informal explanations. On such a basis, it is also possible to specify selected important conditions formally, which have to be enforced as part of the contract. Examples are invariants regarding security conditions or mission-critical consistency conditions. So formal rigour can be applied just to a few selected aspects of the domain.

3.3 Thesis 3: Specification of execution semantics is too much.

In the literature, some approaches can be found which map component architectures and concrete components into formal specifications. The focus is here on concurrent processes and event processing. This kind of work is of course very important in order to develop an overall understanding of the semantic concepts in various programming paradigms. Nevertheless, it is not directly helpful for the work of a component developer or a software developer using components. These developers usually have a quite clear understanding of the operational semantics of the component architecture they are working in. The problem is not so much to define the semantics of the component architecture but to understand the co-ordination and co-operation issues in a given component configuration.

3.4 Thesis 4: Critical component co-ordination issues require precise specifications.

The last two theses left the impression that there is not much to be achieved by formal specification techniques, which is of practical relevance. However, the demand for precise specifications still exists even when we rule out the aspects mentioned in the preceding two theses. When configuring components to an application, there are many questions of the following kinds:

Questions to be answered about the static component configuration:

- In order to make use of component X, which other components have to exist and how are they to be connected to X?
- When configuring the properties of the component X, which rules have to obeyed in order to keep the component working properly? Which dependencies exist to property values of other (connected) components?

Questions to be answered at design time, but related to effects appearing at run time:

- In order to invoke some operation of a component, which preconditions have to be ensured?
- When having executed an operation of a component, which changes to the preconditions of other operations can be inferred?

In general, questions about the right *configuration* of components are important for practical usage of components. So a contract for a component may be rather sloppy on the specification of the actual algorithms contained in the component (as long as these algorithms are well known and do not need further explanation). But a contract has to be very explicit about the constraints that have to be obeyed in the configuration of components and in interconnecting the components.

More generally speaking, there are different aspects of a specification for a business domain. One aspect is the detailed description of all details and functions, which was considered less important in thesis 2 above. Another aspect is the system of rules governing the co-ordination and configuration of the components. Component coordination languages are also subject of recent related research work (see e.g.[MS00] from this workshop).

It is important to see that precise specification of configuration and co-ordination of course requires *some* degree of precise specification of the business domain and of the component runtime semantics (which looks like a contradiction to theses 2 and 3 above). However, it is sufficient to provide these specifications on a much simpler abstraction level where much of the detail information is left out. This abstraction saves time and effort, and contributes to economic viability.

3.5 Thesis 5: Non-functional requirements are important.

The arguments above were essentially restricted to the functional requirements for a software system. In fact, there are several other dimensions of requirements that have to be taken into account in specification of software components.

The purely functional aspect of component specification (which effect is achieved when I invoke some operation?) can be separated into two aspects, as it was indicated above in thesis 2: "Functional essence" of the application domain, which is in many cases of limited relevance, and selected, particularly critical aspects.

Besides the functional aspects, several *non-functional* aspects are relevant for the practical usability of a component:

- What are the resources needed by the component (e.g. memory, hardware/software platform)?
- What is the (average, maximum) response time of the component to an input event?
- Which error rate is guaranteed for the component (based on a classification of error types)?
- Which security/confidentiality properties are guaranteed by the component (e.g. encryption and authentication in communication with other components)?

The above-mentioned aspects of a component belong to a practically useful component specification. The specification has the form of a rely/guarantee contract, i.e. depending on the non-functional properties of the container a component is allocated to. A close integration with the (critical) functional aspects is possible (e.g. performance or confidentiality dependent in individual parts of the functionality).

In the remainder of this text, we will not stress further the issues of non-functional requirements, in order to keep to a limited scope.

3.6 Thesis 6: Formal specifications may provide a flexible mechanism for browsing/viewing component configurations.

When considering the situation of a developer who is configuring a concrete set of components to create an application, and assuming that the components are specified along the lines sketched above, the following opportunities appear:

- Some of the formally specified conditions on component configurations can be checked directly (e.g. presence of some required other components), and feedback can be given immediately to the developer.
- For many other conditions, general consistency rules (like runtime preconditions and invariants) can be instantiated according to the current component configuration. If a formal language with a precise logical semantics is used, the

instantiated rules can be simplified automatically. These more specific versions of the rules are easier to understand for a developer.

- Since components are interconnected in a complex way, the investigation e.g. of a precondition of an operation may span over several components. So the instantiation, composition and simplification of formal component specifications leads to a dynamic creation of textual explanations for the current component configuration, based on some developer-given query. The idea is here to provide a "semantics-directed browser" of the current component configuration.

3.7 Thesis 7: Formal specifications shall be interpreted by machines and still be readable for human beings.

This last thesis is of a very general nature. As it can be seen from the relatively well-functioning system of laws and justice, natural language is in most cases sufficient for establishing contracts. The advantage of using a mathematically precise specification language for contracts is that the specification becomes machine-processable. So it is possible to monitor at runtime whether the contract is fulfilled, it is possible to use sophisticated browsers as mentioned above and it should be possible to automatically create appropriate tests whether a component fulfils its contract.

4 Ideas for Tool Support

4.1 Tool Functionality

Basically, advanced tool support based on formal specifications should be integrated into support tools that follow the state of the art of component-based development. This means that components are represented graphically and special browsers (inspectors) are used to deal with the formal specification of a component in the same way as normal properties are adjusted. So the formal specifications are stored as local parts of the component and packaged together with the component.

Typical examples for formal component constraints in the sense of the discussion above may be, in the context of an order processing system:

- An *OrderProcessing* component assumes that its local properties *customerManagement* and *productManagement* are set to defined components (of the correct type).
- The *CustomerManagement* component can only deal with *Customer* components which have a *customerNumber* and *customerStatus* attribute.
- A precondition for the local method *createOrder* of the *OrderProcessing* component is that the *customerManagement* has checked the respective customer status (whether he/she pays his /her bills, for instance).

These examples show that besides constraints rooted in the business domain, there are also many constraints that are of a more syntactical nature and therefore can be checked mechanically. A support tool can help the developer in many different ways here:

- By documenting and maintaining the constraints (which helps to understand the configuration rules);
- By instantiating the constraints according to the current configuration of component instances and applying static simplifications; (For instance, the constraint that the *customerManagement* property is defined can be removed as soon as it is fulfilled in the current configuration.);
- As a generalisation of the above-mentioned functionality, by providing a flexible browser for the interdependencies among the components;

- By providing an intelligent help function in resolving open issues regarding the component configuration; (e.g. providing a checklist of unresolved constraints);
- By compiling those constraints, which cannot be resolved statically, into dynamic runtime checks.

So the overall functionality envisaged here is not related to verification or any other advanced use of logical calculi. Instead, the focus is on such constraints, which can mechanically evaluated during the application design and testing process. According to the theses from above, it is this kind of support which is most helpful for the developer.

4.2 Specification Language

For the design of support tools, an important question is the concrete choice of the specification language. A language is need with the following properties:

- easy to understand and learn for software developers;
- well integrated with the object-oriented paradigm underlying component technology;
- compatible with object-oriented business domain models, described e.g. in UML;
- compatible with the emerging Component IDL for CORBA;
- applicable in the level of business models as well as on the level of meta-models (for description of component configurations);
- fully machine-executable.

A suitable starting point for the choice of a language with these properties is the Object Constraint Language (OCL) [WK99], which is part of the UML. OCL is able to express various kinds of constraints on objects, including constraints on meta-level (using a reflection mechanism). OCL is explicitly designed for fully automatic mechanical checking of the constraints. First tools exist which evaluate OCL expressions at object configuration time and at runtime [HDF00, RG00].

5 Conclusion and Outlook

In this position statement, it has been argued that there is a clear need for precise semantic component specification. However, a specific approach has been suggested which carefully distinguishes between separate aspects of component specification. The suggested approach does not aim at a general specification of the application domain for a set of components, but tries to give support for understanding the complex interactions of components. It has been stressed that non-functional properties are as least as relevant as functional properties for components.

Some of the ideas mentioned in this paper are currently explored further in research projects at Dresden University of Technology. So an initiative exists for investigating non-functional aspects of components in a co-operation of several computer science disciplines. Moreover, first building blocks for tool support using OCL exists already, in the form of an OCL parser, typechecker and OCL-to-Java compiler [DOCL00]

Acknowledgement: I would like to thank Klaus Bergner, 4Soft München, for interesting email discussions.

References

[CC99] OMG, CORBA Components, Joint revised submission, OMG TC document orbos/99-02-05, 1999

[Cox90] B. Cox, Planning the software industrial revolution. *IEEE Software* (7)6, November 1990.

[Digre98] T. Digre, Business component architecture, *IEEE Software* (15)5, September/October 1998, 60-69.

[DOCL00] F. Finger, B. Demuth, H. Hussmann, Dresden OCL Compiler, see http://www-st.inf.tu-dresden.de/ocl

[HDF00] H. Hussmann; B. Demuth, F. Finger: Modular Architecture for a Toolset Supporting OCL, to appear in Proceedings <<UML>>2000, Conference, October 3-6, 2000, York, UK

[JavaBeans97] Sun Microsystems, JavaBeans specification, http://java.sun.com/beans/doc/spec.htm

[McIlroy68] M. D. McIlroy, Mass produced software components, Proc. Nato Software Eng. Conf., Garmisch, Germany (1968) 138-155.

[Meyer97] B. Meyer, The next software breakthrough, *IEEE Software* (30)7, July 1997, 113-114.

[Monson-Haefel99] R. Monson-Haefel, Enterprise JavaBeans, O'Reilly 1999.

[MS00] C. Montangero, L. Semini, Specification and composition of software components: formal methods meet standards, Proc. Monterey Workshop 2000, part of this volume.

[Pree 97] W. Pree, Komponentenbasierte Softwareentwicklung mit Frameworks, dpunkt 1997.

[RG00] M. Richters, M. Gogolla, Validating UML Models and OCL Constraints, to appear in Proceedings <<UML>>2000, Conference, October 3-6, 2000, York, UK.

[Szyperski97] C. Szyperski, Component software, Addison-Wesley 1997.

[TCI] The Trusted Components Initiative, see http://www.trusted-components.org

[WK99] J. Warmer, A. Kleppe, The Object Constraint Language, Addison-Wesley 1999.

On the Analysis of Dynamic Properties in Component-Based Programming

Paola Inverardi[1] and Alexander L. Wolf[2]

[1] Dip. di Matematica, Universit'a dell' Aquila, I-67010, L'Aquila, Italy
[2] Dep. of Computer Science, University of Colorado, Boulder, Colorado USA

In recent years important changes have taken place in the way we produce software artifacts. The emerging market of commercial off-the-shelf (COTS) components and the increasing spread of component integration technologies such as CORBA, Java/RMI, and COM are determining a completely new way of building distributed systems. Although integration technologies and development techniques assume rather simple architectural contexts (usually distributed, with simple interaction capabilities), they face a critical problem that poses a challenging research issue: understanding if system components correctly integrate.
Component assembly can result in architectural mismatches when trying to integrate components with incompatible interaction behavior, leading to system deadlocks, livelocks, or failing to satisfy desired general functional and non-functional system properties. The approach we present in this paper is focused on preventing and detecting dynamic integration errors in a component-based development setting. We describe a method for deadlock detection that takes a novel approach based on component assumptions and provides a conservative checking algorithm with a state-space complexity significantly lower than comparable approaches. Although the focus in the present paper is on deadlock freedom of fully synchronized components, we believe the method can be generalized to support other composition mechanisms such as asynchronous communication and other properties such as general liveness and safety properties.

1 Introduction

In recent years important changes have taken place in the way we produce software artifacts. On one side, software production is becoming more and more involved with distributed applications running on heterogeneous networks. On the other, emerging technologies such as commercial off-the-shelf (COTS) products are becoming a market reality for rapid and cheaper system development [17]. Although these trends may seem independent, they actually have been bound together with the widespread use of component integration technologies such as CORBA, Java/RMI, and DCOM. Distributed applications are being designed as sets of autonomous, decoupled components, allowing rapid development based on integration of COTS and simplifying architectural changes required to cope with the dynamics of the underlying environment.

Although integration technologies and development techniques assume rather simple architectural contexts, usually distributed, with simple interaction capabilities, they face a critical problem that poses a challenging research issue: understanding whether system components integrate correctly.

There is a growing interest on this topic, both in industrial and military contexts. For example, consider this quote from a recent US Defense Department briefing:

> *"A major theme of this year's demonstrations is the ability to build software systems by composing components, and do it reliably and predictably. We want to use the right components to do the job. We want to put them together so the system doesn't deadlock."*[1]

While for type integration and interface checking, type and subtyping theories play an important role in preventing and detecting some integration errors, interaction properties remain problematic. Component assembly can result in architectural mismatches when trying to integrate components with incompatible interaction behavior [1, 5], resulting in system deadlocks, livelocks, or failing to satisfy desired general functional and non-funtional system properties.

The work we present in this paper represents one step in our general goal of preventing and detecting dynamic integration errors in component-based development. Within this setting, our aim is twofold. First we want to analyze systems in a component-wise manner, that is, we wish to use information providable at a component level to verify system properties. This makes sense in a component-based world: components are bought as is, so if we understand what information component makers should provide, better systems (i.e., ones that have the desired global system properties) can be constructed. Second, we want to be able to verify dynamic properties. This means providing tractable methods that can manage real scale applications even at the cost of completeness. This is what at present type checking provides for a whole class of static correctness properties.

So far, existing techniques for detecting dynamic integration errors are based on behavioral analysis [3, 11] of the composed system model. The analysis is carried on at the system level, possibly in a compositional fashion [6], and has serious problems with state explosion. Our approach [9, 8] is based on enriching component semantics with additional information and performing analysis at a component level without building the system model. Additional information is provided by means of *assumptions*, which are the requirements a component puts on its environment in order to guarantee a certain property in a specific composition context.

The method we formulate starts off with a set of components to be integrated, a composition mechanism (e.g., full sychronization), and a property to be verified (e.g., deadlock freedom). We represent each component with an actual behavior graph (AC). An assumption graph (AS) for proving deadlock freedom is derived

[1] http://www.dyncorp-is.com/darpa/meetings/edcs99jun/

from each AC graph. Our checking algorithm processes all AC and AS graphs trying to verify if the AC graphs provide the requirements modeled by all the AS graphs. The algorithm works by finding pairs of AC and AS graphs that match through a suitable partial equivalence relation. According to the match found, arcs of the AS graph that have been provided for (covered arcs) are marked, and root nodes of both AC and AS graphs are updated. The algorithm repeats this process until all arcs of all AS graphs have been covered or no matching pair of graphs can be found. The former implies deadlock freedom of the system while the latter means that the algorithm cannot prove system deadlock freedom. Consequently, our algorithm is not complete (i.e., there are deadlock free systems that the algorithm fails to recognize), which is the price we must pay for tractability.

Summarizing, the contributions of our approach are a broader notion of component semantics based on assumptions and a method for proving deadlock freedom in a component based setting that is vey efficient in terms of space complexity. While the space complexity of our approach is polynomial, existing approaches have exponential orders of magnitude. In this paper we give an informal description of the steps of our approach and we illustrate the method on a simple example. Complete presentations of the approach in the scope of two different specification contexts, namely CHAM and CCS, can be found elsewhere [8, 9].

2 Related Work

In order to obtain efficient verification mechanisms in terms of space complexity, there has been much effort to avoid the state explosion problem. There are two approaches: compositional verification and minimizations. The first class verifies properties of the individual components, and properties of the global system are deduced from these [12, 14, 18]. However, as pointed out by Grumberg and Long [14], when verifying properties of the components it may also be necessary to make assumptions about the environment, and the size of these assumptions is not fixed. Our approach shares the same motivation but it verifies properties of the component context, represented as fixed size AS graphs, in order to ensure a global system property.

The compositional minimization approach is based on constructing a minimal semantically equivalent representation of the global system. This is managed by successive refinements and use of constraints and interface specifications [6, 7]. However, these approaches still construct some kind of global system representation, and therefore are subject to state explosion in the worst case.

Binary Decision Diagrams [2] are used in many implementations for coding system states. Although it has been proved to be an efficient approach in many cases, it still suffers from space complexity problems.

From the perspective of property checking in large software systems, work in the area of module interconnection and software architecture languages can be mentioned, however the focus is not on efficient property verification of dynamic properties nor is the specific setting of component-based programming taken into account [9].

There are other attempts at proving partial deadlock freedom statically. Kobayashi and Sumii [10, 16] propose a type system that ensures certain kinds of deadlock freedom through static checking. Their approach is based on including the order of channel use in the type information and requiring the designer to annotate communication channels as reliable or unreliable. As in our work, they use behavioral information to enhance the type system, however part of the additional information must be provided by users and is related to channels rather than components. In our approach, additional information is derived from the property to be proved and the communication context. Besides, the derived information extends component semantics, thus integrating well with the current direction that software development has taken, based on component integration technologies and commercial off-the-shelf products.

3 Property Checking Using Assumptions

We represent component behavior (and component assumptions later on) using directed, rooted graphs. We define the notion of ***actual behavior*** (AC) graph for modeling component behavior. The term "actual" emphasizes the difference between component behavior and the intended, or assumed, behavior of the environment. AC graphs model components in an intuitive way. Each node represents a state of the component and the root node represents its initial state. Each arc represents the possible transition into a new state where the transition label is the action performed by the component.

In this section we present the various steps upon which our approach is based, i.e., how component assumptions can be derived and used for proving deadlock freedom in a system composed of a finite number of components that communicate synchronously. Following a common hypothesis in automated checking of properties of complex systems [11], behavior of all components can be finitely represented.

3.1 Deriving Assumptions for Deadlock Freedom

We wish to derive from a component behavior the requirements on its environment that guarantee deadlock freedom. A system is in deadlock when it cannot perform any computation, thus in our setting, deadlock means that all components are blocked waiting for an action from the environment that is not possible. Our approach is to verify that no components under any circumstance will block. This conservative approach suffices to prove deadlock freedom exclusively from component assumptions. The payback is efficiency, while the drawback is incompleteness.

Let us consider a context in which components are combined together, composing them in parallel and forcing them to synchronize whenever possible, where synchronization is obtained when they offer at the same time complementary actions [8]. In this context, a component will not block if its environment can always provide the actions it requires for changing state. Thus, we can define the notion of component assumption in the context of parallel composition and deadlock freedom as a sort of complementary graph of the AC graph, that is, a graph that is structurally identical to the AC graph but whose labels are complementary with respect to the corresponding labels in the AC graph. We call this graph the *assumption graph* (AS).

3.2 Checking Assumptions

Once component assumptions have been derived, we wish to verify if these assumptions are satisfied by the environment, which, intuitively, is the rest of the components in the given context. This satisfaction relation reduces to proving if the component environment is equivalent to the component assumption by means of a suitable notion of equivalence. The idea behind the definition of equivalence we use is that the graphs can always imitate each other. If a graph performs an action l, the other graph can also perform l and, no matter what internal choices it may make, it will be able to continue imitating the other graph. However, our notion of equivalence is more restrictive than the notion of weak bisimilarity [13], since we need to assure that a given behavior *must* be provided by all the branches that provide the matched portion.

We verify the equivalences between AS graphs and environments without constructing the whole environment behavior. The main idea is to allow a portion of a component behavior to provide a portion of another component assumption. For this we need to provide a notion of *partial equivalence* that preserves equivalence in a conservative way. Once a partial equivalence has been established, the assumption graph has been satisfied to some extent and therefore some marking mechanism is necessary in order to record it.

Partial Equivalence A partial equivalence between an AC and an AS graph allows the equivalence relation to be defined up to a certain point in the graphs. The AC and AS graphs are not required to be completely equivalent; their root nodes must be equivalent, the nodes reachable from root nodes too, and so on until a set of nodes called *stopping nodes* is reached. Stopping nodes represent the points where the actual behavior will stop providing the assumption's requirements, hence there should be another AC graph capable of doing so from then on.

Checking Algorithm and an Example Application The checking algorithm is very simple. It iteratively finds partial equivalencies between AC and AS graphs, marks all the fulfilled assumptions, and changes the roots of both graphs. Iteration stops when all assumptions are completely marked. An important point is that partial equivalences guarantee that the matched portions of assumptions cannot be matched in any other way, therefore the order in which partial matches are applied does not affect the correctness of the algorithm.

We now apply the algorithm to the Compressing Proxy example [4,9]. To improve the performance of UNIX-based World Wide Web browsers over slow networks, one could create an HTTP (Hyper Text Transfer Protocol) server that compresses and uncompresses data that it sends across the network. This is the purpose of the Compressing Proxy, which weds the **gzip** compression/decompression program to the standard HTTP server available from CERN.

The main difficulty that arises in the Compressing Proxy system is the correct integration of existing components. The CERN HTTP server consists of *filters* strung together in series executing in one single process, while the **gzip** program runs in a separate UNIX process. Therefore, an adaptor must be created to coordinate these components correctly (see Figure 1).

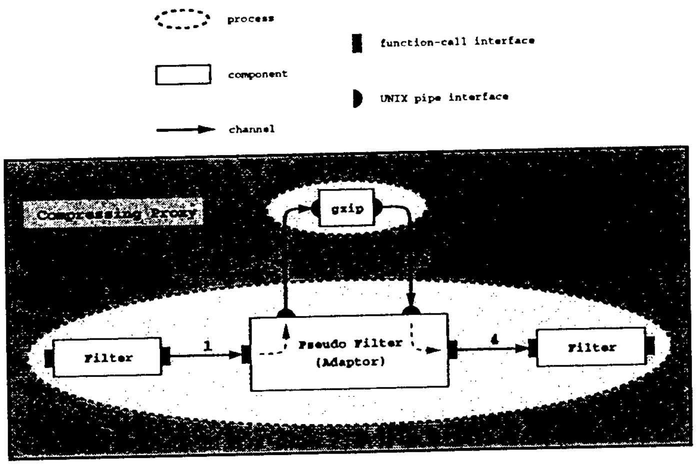


Fig. 1. The Compressing Proxy.

However, the correct construction of the adaptor requires a deep understanding of the other components. Suppose the adaptor simply passes data on to **gzip** whenever it receives data from the upstream filter. Once the stream is closed by the upstream filter (i.e., there are no more data to be compressed), the adaptor reads the compressed data from **gzip** and pushes the data toward the downstream filter.

At a component level, this behavior makes sense. But at a global system level we can experience deadlock. In particular, **gzip** uses a one-pass compression algorithm and may attempt to write a portion of the compressed data (perhaps because an internal buffer is full) before the adaptor is ready, thus blocking.

With **gzip** blocked, the adaptor also becomes blocked when it attempts to pass on more of the data to **gzip**, leaving the system in deadlock.

A way to avoid deadlock in this situation is to have the adaptor handle the data incrementally and use non-blocking reads and writes. This would allow the adaptor to read some data from **gzip** when its attempt to write data to **gzip** is blocked.

We represent partial equivalences with dotted lines for related nodes and crosses for stopping nodes. In Figure 2, the upstream filter matches successfully with the adaptor. Once the successful match has been made, both graphs are modified. The new state of the adaptor can be seen in Figure 3.

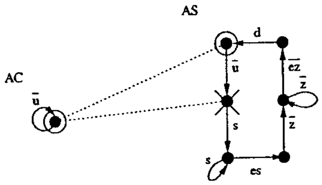


Fig. 2. Successful Match of Upstream Filter AC Graph against Adaptor AS Graph.

Figure 3 shows how a partial match can be established between the **gzip** AC graph and the adaptor AS graph. However, it is possible to see that there is no way of extending the relation in order to cover the edge labeled $\overline{es}$. Hence, the algorithm, after all possible attempts, terminates, indicating that the proposed configuration is presumably not deadlock free.

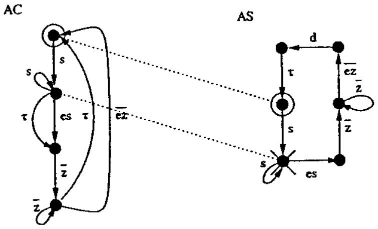


Fig. 3. Unsuccessful Match of **gzip** AC Graph against Adaptor AS Graph.

Notice that the mismatch occurs precisely where the deadlock in the system appears: **gzip** may attempt to output the compressed file ($\bar{z}$) while the adaptor is expecting to be synchronizing with a component, inputting an *end of source* (*es*) before the compressed file is output.

The adaptor must be modified to prevent system deadlock. In Figure 4, the partial equivalence that covers the $\overline{es}$ edge allows the modified adaptor's AS graph to be updated, and Figure 5 finishes covering the AS graph completely. The algorithm goes on matching AC and AS graphs until all arcs of all AS graphs are covered. Thus, the checking algorithm finally returns true, meaning that the proposed system is deadlock free.

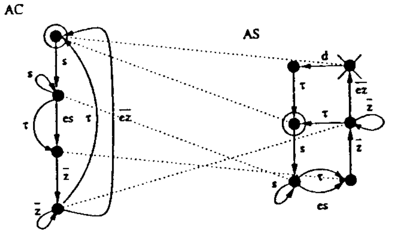


Fig. 4. Successful Match of **gzip** AC Graph against a Modified Adaptor AS Graph.

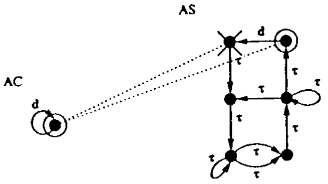


Fig. 5. Successful Match of Downstream Filter AC Graph against a Modified Adaptor AS Graph.

4 Method Assessment

Up to now we have informally introduced a method for checking deadlock freedom that trades off completeness for efficiency. We now briefly comment on the completeness and complexity of our approach. A detailed discussion of all the hypothesis and implications on the complexity, completeness, and correctness of our approach can be found elsewhere [8].

4.1 Complexity

The algorithm we sketched above offers a partial solution to the state explosion problem. In our approach, deadlock freedom is proven without building the entire finite-state model of the system. We only construct finite representations of each component individually: an actual behavior graph and an assumed behavior graph of its context.

In standard aproaches, using reachability analysis, the complete state space of the system is built. If we consider a concurrent system composed of N components of comparable size, whose finite state representation is of size $O(K)$, then the composed system state space is $O(K^N)$. Although there are many techniques for reducing the state space, such as automata minimization and "on the fly" algorithms, the worst case still requires the whole state space to be analyzed, leading to a time complexity of $O(K^N)$.

In our approach only two copies of each component are built, AC and AS graphs. Thus, following the same considerations as before, the state space complexity is radically improved to $O(KN)$. On the other hand, in terms of time complexity, the worst case of our algorithm is $O(N^3K^4log(K))$, which is comparable to the worst case of standard reachability. The time complexity results from the following: Establishing a partial equivalence relation between two graphs can be considered a variation of the standard bisimulation checking. Thus, the upper bound on its complexity would be $O(K^2log(K))$ [15]. However, the partial equivalence must be established for a pair of graphs. Thus, all possible pairs must be checked $(Comb(N,2))$, leading us to $O(N^2(K^2log(K)))$. Finally, considering the worst case in which each partial match only covers a single arc of the NK^2 possibilities. We get $O(K^2N^3(K^2log(K)))$, which reduces to $O(N^3K^4log(K))$.

4.2 Completeness

The approach presented in this paper may be considered incomplete in two different ways: firstly, some restrictions on the systems for which the method can be used are necessary, and secondly, because the checking algorithm may not be able to conclude deadlock freedom for some deadlock-free systems.

The first restriction on the system requires components to be able to perform each computation an infinite number of times, but does not affect the completeness of out approach. The goal of this restriction is only for the sake of simplicity of the formal presentation. The second restriction is more serious. We do not accept that more than two components have shared channels. If a communication channel can be used by more than two components, there is a potential nondeterminism in the overall system behavior. A component may have the possibility of synchronizing with one of several components leading to a nondeterministic choice. In terms of our approach, this means that one cannot commit to which AC graph will provide the AS graph requirements. As the matching process guarantees that the matched arcs of the AS graph will always be provided by the AC graph, no matching can be done. The nondeterminism introduced by shared channels is similar to the nondeterminism that makes our algorithm incomplete.

Having discussed the restriction imposed on components, the incompleteness of the checking algorithm remains. Our approach is intrinsically incomplete. First of all, we attempt to prove a global property such as deadlock in terms of local properties of each component. Second, we verify equivalences between the context and component assumptions in sucessive partial steps so as to not construct a complete model of the component context. As a consequence, the characteristics of our setting lead to the following situation: Given a deadlock-free system, the algorithm may not be able to conclude that it is deadlock free. What happens is that the algorithm reaches a state in which it cannot do further matches between AC and AS graphs. However, the main reason for the incompleteness of our approach is nondeterminism. When there is a nondeterministic choice in a component's behavior, when a component can interact with one of two different components, there cannot be a unique mathing that guarantees how the system will evolve. In these situations the algorithm stops without obtaining AS graphs completely matched, and therefore not giving a conclusive answer. Incompleteness is the price that must be paid to make analysis tractable. Our method may apply only to a subset of problems, but it lowers the complexity of the solution from exponential order to a polynomial one.

5 Conclusions and Future Work

In this paper we have informally illustrated a preliminary space-efficient approach to proving dynamic properties of component-based systems. The approach assumes a broader notion of component semantics based on assumptions and a method for proving deadlock freedom in a component-based setting. This method is based on deriving assumptions (component requirements on its environment in order to guarantee a certain property in a specific composition context) and checking that all assumptions are guaranteed through a partial matching mechanism. The method is considerably more efficient than methods based on system model behavior analysis, since its space complexity is polynomial, while existing approaches have exponential orders of magnitude. It is not complete, but it allows the treatment of systems whose synchronization patterns are not trivial.

Ongoing and future work is proceeding in two directions. First, to validate the proposed framework through experimental results, we are currently working on an implementation of the algorithm, and considering other coordination contexts, such as non-fully synchronized or asynchronous ones. Second, to extend the approach to deal with other properties, such as general liveness and safety properties, we are thinking of general safety properties expressed with property automata, such as in the Gas Pump example [11] that may be decomposed into component assumptions or specific component assumptions such as particular access protocols for shared resources.

We believe that assumptions are a good way to extend component semantics in order to verify properties more efficiently. The approach presented in this paper is an example of how this can be achieved.

Acknowledgements

We would like to thank Sebastián Uchitel and Daniel Yankelevich for discussions and common work on the subject of the paper.

References

1. B. Boehm and C. Abts. COTS Integration: Plug and Pray? *IEEE Computer*, 32(1), January 1999.
2. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.J. Hwang. Symbolic model checking : 10^{20} and beyond. *Information and Computation*, 98:142–170, June 1992.
3. R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: a semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.
4. D. Compare, P. Inverardi, and A.L. Wolf. Uncovering Architectural Mismatch in Dynamic Behavior. *Science of Computer Programming*, 2(33):101–131, February 1999.
5. D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch: Why Reuse is so Hard. *IEEE Software*, 12(6), November 1995.
6. D. Giannakopoulou, J. Kramer, and S.C. Cheung. Analysing the Behaviour of Distributed Systems using Tracta. *Automated Software Engineering, special issue on Automated Analysis of Software*, 6(1):7–35, January 1999.
7. S. Graf, B. Steffen, and G. Lüttgen. Compositional minimisation of finite state systems using interface specifications. *Formal Aspects of Computing*, 8(5), 1998.
8. P. Inverardi and S. Uchitel. Proving Deadlock Freedom in Component-Based Programming. Technical report, Universita' dell'Aquila, Italy, November 1999.
9. P. Inverardi, A.L. Wolf, and D. Yankelevich. Static Checking of System Behaviors Using Derived Component Assumptions. *ACM Transactions on Software Engineering and Methodology*, 2000. To appear.
10. N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20(2):436–482, March 1998.
11. J. Kramer and J.C. Cheung. Compositional reachability analysis of finite-state distributed systems with user-specified constraints. In *SIGSOFT95: 3rd International Symposium on the Foundations of Software Engineering*, pages 140–150, Washington D.C., October 1995.
12. K. Laster and O. Grumberg. Modular model checking of software. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, Lisbon, March 1998.
13. R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.
14. O.Grumberg and D.E.Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.
15. R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.
16. E. Sumii and N. Kobayashi. A generalized deadlock-free process calculus. In *3rd International Workshop on Hish-Level Concurrent Languages*, volume 16 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 1998.
17. Clemens Szyperski. *Component Software. Beyond Object Oriented Programming.* Addison Wesley, Harlow, England, 1998.
18. G. Winskel. Compositional checking of validity on finite state processes. In *Workshop on Theories of Communication, CONCUR*, volume 458 of *LNCS*, 1990.

A Formal Model of System and Software Engineering Experience

Douglas S. Lange
SPAWARSYSCEN
D44207
53118 GATCHELL RD RM 426C
San Diego, CA 92152-7420
dlange@computer.org

Valdis Berzins
Naval Postgraduate School
Computer Science Dept. Code CS/Bv
833 Dyer Road
Monterey, CA 93943-5118
berzins@cs.nps.navy.mil

Abstract

This paper describes some of the issues involved in formalizing a previously fuzzy aspect of software development – the use of experience. This paper reports the status of work in progress and raises some issues for thought.

Introduction and Motivation

Software and systems engineering organizations need to improve their performance from project to project. Such improvement can only come about from understanding and analyzing the experience gained during past projects. Since organizations often outlast the membership of individuals within that organization, and since collaboration among many individuals is often required, automated storage, retrieval, and decision support should be useful. Efforts to develop engineering experience bases in the literature focus on databases of artifacts, preserving free text correspondence, and developing structured metadata for lessons learned documents. Our position is that these are not sufficient models for an engineering experience base because they do not capture the semantics of the experiences at a level that can be effectively used by decision support algorithms.

The field of artificial intelligence has provided approaches that can be used to associate the conditions that led to previous decisions, the decisions themselves, and the measured and analyzed results. Similarly through applications of modal logic, elements of experience that seemingly contradict each other can coexist in an experience base and provide useful insight to decision makers. We propose that a more formal logic-based model is essential in order to build an experience base that can provide substantive computer aid to support engineering process improvement.

Background

This section surveys and assesses relevant previous work.

Organizational Memory

Organizations can be seen as functionally resembling information-processing systems that process information from the environment. As such, organizations exhibit memory that is similar in function to that of individuals. The components of the system include: [WalUng91]

- Sensors, which act to receive information
- A processing capability that processes information using defined symbols
- A memory where information is sent for storage and from which it can be retrieved.

Individuals in problem-solving and decision-making activities acquire information. Organizational memory comes about due to the sharing of information among individuals in different ways. Organizational memory is stored information about a decision stimulus and response from an organization's history that the organization can use to help make present decisions. [WalUng91] This definition points out a critical factor in defining organizational memory that has been missed by experience base researchers. Without information about the decision stimulus, the "why" of an organizational response cannot be determined. Walsh and Ungson argue that only if the decision stimulus is retained can the experience be used to meet the requirements of more novel situations, and that without it one is likely to promote deleterious decision making.

Continuous Process Improvement

IEEE 1220 is among the standards available for system engineering concerns to use in defining their engineering process. According to the standard, within the policies and procedures of the project, the continuous improvement of products and processes must be addressed. Among the activities recommended by the standard for accomplishing continuous improvement of processes are:

- Maintaining a self-assessment program to determine the maturity of the enterprises systems engineering practices, and
- Capturing the lessons learned on each project and incorporate them into enterprise training courses, as appropriate, to improve the application of the SEP.

IEEE 1220 goes on to state that in order to be compliant and perform systems engineering at a standard industry level, one must have a means to capture the experiences generated by undertaking projects. [IEEE 1220] The issue we wish to explore is what kinds of formal models would be appropriate for that purpose.

Experience Bases

Software and systems engineering experience base research can be divided into three categories. First are previous efforts to improve the use of largely unstructured lessons learned artifacts. A second category uses the tools available in free-text applications such as email and chat capabilities. Third are database applications. These are either based on relational or object databases.

Lessons Learned

Lessons learned documents are among the least structured objects in which experience can be preserved. It is not surprising therefore that research in improving the management of lessons learned documents would seek to impose some structure on them. Birk and Tautz describe a process for packaging lessons learned that has as its basis a quality control mechanism and a structuring of the document [BirTau98]. The structure used divides a lesson-learned document into four sections.

Object — The software artifact that the lesson is concerned with. It can be a tangible product, or something less tangible such as a process or method.

Context — Describes the situation within which the problem and solution are relevant.

Problem — The problem that is being solved. This along with the solution is the core of the lesson learned.

Solution — The solution to the problem. This along with the problem is the core of the lesson learned.

Birk and Tautz go on to classify types of lessons learned through a semantic network of presupposed relationships among the types of lessons learned and their relationship to situations and artifacts. Other studies support the importance of classifying lessons in order to impose a structure that allows lessons to be retrieved. Statz creates elements for lessons, each of which can have multiple attributes [Sta99]. These form the equivalent of facets for searching a lessons learned database.

Free-Text Applications

There is no faster, less intrusive way to capture experience than to record the conversations of the people involved in an effort. At its simplest, this approach can take on the form of a running logbook. An example of this approach uses a distributed running diary as the experience base. The diary serves to allow process participants to record problems and solutions in free text form. Search facilities allow users to go back and look at previous entries [Rob+00].

Another approach to capturing free text is found in the Answer Garden [Ack98]. Answer Garden attempts to augment organizational knowledge in two ways. First, Answer Garden makes answers to common questions available through structured and unstructured user interactions. Second, it provides information about who in an organization is considered an expert in a particular area. Answer Garden provides a hierarchical structure to help users find the answers to frequently asked questions. The user interface asks a series of questions. The answers drive the user along a tree structure towards an answer. If the problem to be solved is new and an answer cannot be found, the application prompts the user to describe the problem and it is sent to an expert. The expert answers the question and can then place it and the answer into the hierarchy so that future users can find it. Free text search engines also allow the user to find answers directly.

Database Applications

Databases provide a natural platform on which to store the experiences of an organization. When designing an experience base using such a tool, the foci of the effort are on what types of information will be collected and how will a user find the information needed to solve a particular problem [Alt+99] [BroRun99] [Wan+99]. The primary deficiency of database applications for experience bases is that they store only artifacts and data. These can either be the results of the project or the results of the packaging of experience in the form of new process definitions. The rationale and the reasoning behind the experience is missing. This makes it difficult for the recipient to generalize and determine whether or not the experience is applicable. The contributions of these research efforts are mostly in the area of information retrieval.

Assessment

The common weakness of all three approaches to experience bases is that they passively store data and rely on the users to perform all interpretation, analysis, and adaptation. Better formal models that can support some aspects of the intended use of this data are needed.

Evolution Control Systems

Evolution control systems provide a hint at a path to provide better formal modeling and decision support. All of the standards mentioned above require that projects keep track of information that allows them to adequately control their processes. Such information includes:

- Requirements,
- Schedules,
- Defects,
- Process descriptions, and
- Process metrics.

The configuration management process [SEI99] requires that a project be able to establish and track baselines, manage changes to baselines, and be able to perform audits tracing changes to and from engineering decisions.

The current state of the practice in this area involves using configuration management tools to help manage this information. Similar practices exist for other process areas. The state of the research involves the use of formalized process definitions that allow more powerful automated evolution control systems to be developed. These systems can form the basis around which tools supporting

As defined in the Relational Hypergraph Model of Software Evolution (RHMSE) software evolution consists of two main sub-processes. First is the software prototype evolution process and second is a software production generation process [Har+99]. The figure below illustrates the evolution lifecycle. Throughout the prototyping process, the elements communicated are requirements, software prototypes, demonstrations, and user criticisms [Ber+97].

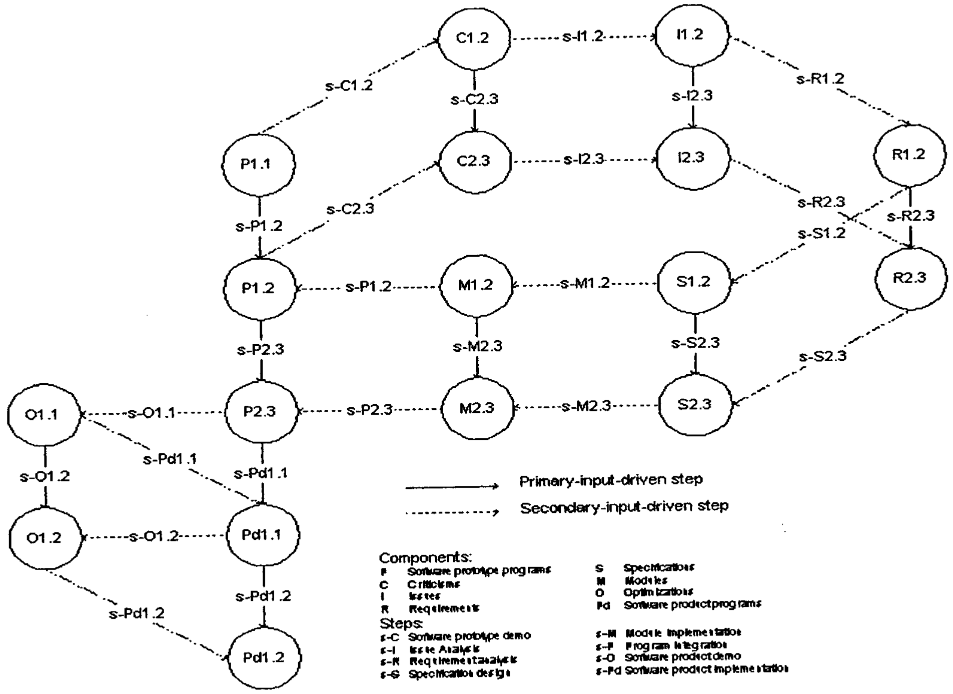


Figure 1. Software Evolution Hypergraph

Several of the elements necessary for experience bases can come directly from an evolution control system. The information tracked as well as the semantic network representation of the evolution history provide context points to which problems and solutions can be attached.

Why a Formal Model?

Process improvement is by nature an act of inference. In order to determine how to improve a process, an analyst must use knowledge of the previously employed processes, information about the environment in which the process has been executed, and the measurements taken during execution. Problems and their solutions figure highly not only in revising documented processes, but also in solving unanticipated problems as they arise. For organizational memory to be used in novel situations, decision makers must be able to *generalize* from previous experience to encompass the current problem, as they perceive it.

Perception too argues against simple collections of artifacts. Not only must the experience base be able to present experience in light of different environments, but the beliefs of the engineers about the environment and the results of their efforts must be modeled. Two engineers may come to different conclusions based on their differing beliefs and these differences can in themselves provide valuable insight.

The ultimate argument for a formal model is to make automation of information collection and knowledge inference possible. Information vital to an experience base is difficult to collect. It is time consuming and does not directly relate to, what most engineers feel, is the job at hand. By representing experience in a formal model, and in particular one that can be represented by a semantic network, a linkage to an evolutionary control system such as the one described above becomes possible.

Expected Characteristics

The following characteristics are expected to drive the form of the model being developed.

- Experience is a collection of beliefs about what has happened in the past and what the situations were that were relevant to the outcomes. Modal operators will be necessary. Modality should allow assumptions of monotonic behavior (within each modality) to be accepted. Without modality, non-monotonic examples are likely.
- More than one point of view may exist about what the situations or results were concerning a previous project. One, both, or neither may be correct. The points of view must be linked to the corresponding supporting evidence to support judgments of which point of view is most relevant to a given problem.
- Generalization cannot be done without the beliefs concerning the environment that existed during previous projects.
- Time may or may not be relevant to the results experienced. One may not know whether it is relevant until attempting to generalize or compare to a current situation. Temporal operators will be necessary.
- Vital components of the experience base will be directly extracted from evolution control constructs. Examples of this are the requirements set, task assignments, personnel qualifications, actual and predicted labor and task duration, test results, demonstration plans, and user feedback in the form of criticisms.
- The use of a semantic network for the model rather than linear logic descriptions will allow easier integration with a human computer interface.
- The use of modal logic will allow easier integration with a standard agent architecture [FIPA].

Issues

The following issues are expected to arise during the model's development.

- Experience can change one's beliefs about past situations. This will likely cause a problem.
- We will need to show that information is entered into the system without onerous effort on the part of the engineers. It will be difficult to decide how much effort is appropriate. It is believed this would vary from organization to organization and that it will therefore be difficult to study. However, statistical data that shows values within two standard deviations might be a useful result.

Example

To examine the rationale behind our belief that a logic-based model of experience is needed, the expected characteristics of the model, and the issues anticipated, the following scenario is analyzed.

Scenario

A test team is putting together a test plan and procedures to support version three of a large software project. Two years of modest fixes and updates occurred between versions one and two, and two years have also occurred since version two was released. The last test plan was written nearly three years ago.

When version one was tested, a large effort was made to exhaustively examine the application programmer interface (API) set of the major server components in the system. Many third party applications as well as large parts of the software being tested make use of these services and it was felt that this was an important risk reduction strategy. When version two was tested, no API testing was conducted.

No members of the version three team were part of the API test effort. Neither were any of them part of the decision processes that resulted in testing the API in version one and not testing them in version two.

A Database Solution with Case Based Reasoning for Information Retrieval

Two test plans are in the experience base. Based on the solution offered by [Wan+99], there is information of the following nature associated with each test plan.

Attribute	Test Plan for Version 1	Test Plan for Version 2
Organization	C2 Systems Engineering	C2 Systems Engineering
Staff size	15	10
Application domain	Command and Control	Command and Control
Improvement goal	Reliability	Reliability
Programming Language	C	C
Software System Size	2000 KLOC	2500 KLOC

Table 1. Attribute Data for Artifacts

Which should be retrieved? Using [Wan+99], this would depend on our goals. Our goal is to create a new test plan. Our current staff size is eight, and the current system size is 3000 KLOC, so version 2 will come out as the closest match. Was this the right decision? There may be an artifact in the system that is a lesson's learned document describing the decision process if somebody wrote one. This would take us to the approach in [BirTau98] with retrieval being case based.

Our Approach

Our approach involves using the information created through the use of an evolution control system. The decisions made are represented in such a system through the assignments created, the criticisms levied against a system, and the issues and requirements managed [Ber+97]. The relationships between objects in such a system need to be expanded however. [Ber+97] limited the relationships to *poses, affects, supports, depends_on*. The relationships between decisions and the supporting information must be recorded

When users began to notice problems with the services, the following criticisms would be posed. They are shown in the form of triples alongside text that describes some of the content of the information nodes.

User1, poses, Criticism1

Criticism1, affects, Version1

User1, poses, Criticism2

Criticism2, affects, Version1

User2, poses, Criticism3

Criticism3, affects, Version1

...

As criticisms are entered, analysts are assigned the task of constructing and associating issues and requirements to the criticisms. The analysts' issues and the relationships they find among them are recorded in the evolution control system. An experience base can use them beyond requirements tracing. Each analyst's entries illustrate something about their beliefs concerning the state of the project. Therefore, when *Analyst1*, decides that the services and their associated API are not well enough tested, she creates an issue and associates all of the criticisms that she believes are symptoms of that problem to the issue. Further analysis indicates that the issue is related to the requirements that created the services in question and their related API.

Criticism1, affects, Issue1

Criticism2, affects, Issue1

...

Criticism1000, affects, Issue1

Issue1, affects, Requirement2

Issue1, affects, Requirement3

...

Further *support* for *Analyst1's* belief can be found in schedule changes recorded in the evolution control system. Schedule slips corresponding to the difficulties that developers are having with the services form part of the rationale for the project manager to create a requirement for full API testing.

StepState223, updates27, StepState156

updates27, reason3, rationale15

updates27, supports3, Issue1

...

The experience base adds decision nodes (among others) to the features tracked by the evolution control system. The decision to create a requirement for API testing follows from the support created by the large set of criticisms all relating to a single issue and by the schedule slips that have been associated to the same issue.

In version two of the system, the defects did not appear, because no new service creation requirements were assigned. Basically reuse of the services with a few repairs was the decision. Therefore no requirement to test the API set in version two was created.

It is now time to plan the testing for version three. New requirements similar to those in version one are created. The experience base agents infer that these new requirements that also include interface components could also support the belief that API testing is an important feature for the test plan. If no new interface requirements were added to the system the inference would not be supported and the agent would not recommend the API testing.

In order for this inference to occur, a basic model of features and relationships of software engineering projects must be present in an ontology service available to the agents. Information such as interface requirements being a special form of requirement will need to be "known" by the agents. The semantics of the relationships will need to be defined in more detail than currently used by evolution control systems, but their current relationships will map into the ontology without affecting their capabilities.

Conclusions

Through a more formal model of software engineering experience, human and automated inference can be improved in support of engineering decision-making. An equally important benefit will be the ability to link the experience base to an evolution control system. In this way, the collection of experience becomes a side effect of the normal management decision process.

References

[Ack98] Ackerman, M., "Augmenting organizational memory: a field study of answer garden", *ACM Transactions on Information Systems*, Vol. 16. Issue 3, 1998.

[Alt+99] Althoff, K., Birk, A., Hartkopf, S., Müller, W., Nick, M., Surmann, D., and Tautz, C., "Managing Software Engineering Experience for Comprehensive Reuse", Proceedings of the Eleventh International Conference on Software Engineering and Knowledge Engineering, Kaiserslautern, Germany, 1999.

[Ber+97] Berzins, V., Ibrahim, O., Luqi: "A Requirements Evolution Model for Computer Aided Prototyping", Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering, Madrid, Spain, 1997.

[BirTau98] Birk, A. and Tautz, C., "Knowledge Management of Software Engineering Lessons Learned", Proceedings of the Tenth International Conference on Software Engineering and Knowledge Engineering, San Francisco Bay, California, USA, 1998.

[BroRun99] Broomé, M. and Runeson, P., "Technical Requirements for the Implementation of an Experience Base", Proceedings of the Eleventh International Conference on Software Engineering and Knowledge Engineering, Kaiserslautern, Germany, 1999.

[FIPA] Foundation for Intelligent Physical Agents. http://www.cselt.stet.it/fipa/.

[Har+99] Harn, M., Berzins, V., and Luqi, "Computer-Aided Software Evolution Based on a Formal Model", Proceedings of the 13th International. Conference on Systems Engineering, Las Vegas, NV, USA, 1999.

[IEEE1220] IEEE Std 1220-1998, *IEEE Standard for Application and Management of the Systems Engineering Process*, Institute of Electrical and Electronics Engineers, 1998.

[Rob+00] Robinson, M., Kovalainen, M., and Auramäki, E., "Diary as Dialogue in Papermill Process Control", *Communications of the ACM*, Vol. 43, No. 1, January 2000.

[SEI99] Software Engineering Institute, Capability Maturity Model®-Integrated-Systems/Software Engineering: Staged Representation - Volume 1, Version 0.2b, 1999.

[Sta99] Statz, J., "Leverage Your Lessons", *IEEE Software*, Vol. 16. No. 2, IEEE, 1999.

[WalUng91] Walsh, J. and Ungson, G., "Organizational Memory", *Academy of Management Review*", Vol. 16., No. 1, January 1991.

[Wan+99] Wangenheim, C., Althoff, K., and Barcia, R., "Intelligent Retrieval of Software Engineering Experienceware", Proceedings of the Eleventh International Conference on Software Engineering and Knowledge Engineering, Kaiserslautern, Germany, 1999.

A Risk Assessment Model for Evolutionary Software Projects[1]

Luqi, J. Nogueira
Naval Postgraduate School
Monterey CA 93943 USA

Abstract

Current early risk assessment techniques rely on subjective human judgments and unrealistic assumptions such as fixed requirements and work breakdown structures. This is a weak approach because different people could arrive at different conclusions from the same scenario even for projects with a stable and well-defined scope, and such projects are rare. This paper introduces a formal model to assess the risk and the duration of software projects automatically, based on objective indicators that can be measured early in the process. The model has been designed to account for significant characteristics of evolutionary software processes, such as requirement complexity, requirement volatility and organizational efficiency. The formal model based on these three indicators estimates the duration and risk of evolutionary software processes. The approach supports (a) automation of risk assessment and, (b) early estimation methods for evolutionary software processes.

1. Introduction

Software applications have grown in size and complexity covering many human activities of importance to society. The report of the President's Information Advisory Committee calls software the "new physical infrastructure of the information age". Unfortunately, the ability to build software has not increased proportionately to demand [Hall, 1997. pp xv], and shortfalls in this regard are a growing concern. According to the Standish group, in 1995 84% of software projects finished over time or budget, and $80 billion - $100 billion is spent annually on cancelled projects in the US. Developing software is still a high-risk activity.

There have been many approaches to improving this situation, mostly focused on increasing productivity via improvements in technology or management. Although better productivity is certainly welcome, closer examination shows that these efforts address only half of the problem. A project gets over time or over budget if actual performance does not match estimates. Current estimation techniques are far from reliable, and tend to systematically produce overly optimistic estimates. More accurate early estimates could help reduce wasted resources associated with overruns and cancelled projects in two ways: if costs are known to be too high at the outset, the scope of the project could be reduced to enable completion within time and budget, or it could be cancelled before it starts, and instead the resources could be used to successfully complete other feasible projects.

This paper therefore focuses on improved risk assessment for software projects. We address project risks related to schedule and budget, and focus mostly on completion time of the project. Current risk assessment standards are weak because they rely on subjective human expertise, assume frozen requirements, or depend on metrics difficult to measure until it is too late. This paper describes a formal risk assessment model based on metrics and sensitive to requirements volatility. Further details can be found in [Nogueira 2000]. The model is specially suited for evolutionary prototyping and incremental software development.

Section 2 defines the problem we are addressing. Section 3 analyzes relevant previous work. Section 4 presents and evaluates our project risk model. Section 5 outlines how systematic risk assessment fits into iterative prototyping. Section 6 concludes.

[1] This research was supported in part by the U. S. Army Research Office under contract/grant number 35037-MA and 40473-MA, and in part by DARPA under contract #99-F759.

2. The Problem

As the range and complexity of computer applications have grown, the cost of software development has become the major expense of computer-based systems [Boehm 1981], [Karolak 1996]. Research shows that in private industry as well as in government environments, schedule and cost overruns are tragically common [Luqi 1989, Jones 1994, Boehm 1981]. Despite improvements in tools and methodologies, there is little evidence of success in improving the process of moving from the concept to the product, and little progress has been made in managing software development projects [Hall, 1997]. Research shows that 45 percent of all the causes for delayed software deliveries are related to organizational issues [vanGenuchten 1991]. A study published by the Standish Group reveals that the number of software projects that fail has dropped from 40% in 1997 to 26% in 1999. However, the percentage of projects with cost and schedule overruns rose from 33% in 1997 to 46% in 1999 [Reel 1999].

Despite the recent improvements introduced in software processes and automated tools, risk assessment for software projects remains an unstructured problem dependent on human expertise [Boehm 1988, Hall 1997]. The acquisition and development communities, both governmental and industrial, lack systematic ways of identifying, communicating and resolving technical uncertainty [SEI 1996].

This paper explores ways to transform risk assessment into a structured problem with systematic solutions. Constructing a model to assess risk based on objectively measurable parameters that can be automatically collected and analyzed is necessary. Solving the risk assessment problem with indicators measured in the early phases would constitute a great benefit to software engineering. In these early phases, changes can be made with the least impact on the budget and schedule. The requirements phase is the crucial stage to assess risk because: a) it involves a huge amount of human interaction and communication that can be misunderstood and can be a source of errors; b) errors introduced at this phase are very expensive to correct if they are discovered late; c) the existence of software generation tools can diminish the errors in the development process if the requirements are correct; and d) requirements evolve introducing changes and maintenance along the whole life cycle.

Part of the problem is misinterpreting the importance of risk management. It is usually and incorrectly viewed as an additional activity layered on the assigned work, or worse, as an outside activity that is not part of the software process [Hall 1997, Karolak 1996]. One of the goals of our research is to integrate a risk assessment model with previous research on CAPS[2] at NPS [Harn 99]. This integration is required in order to capture metrics automatically in the context of a modern evolutionary prototyping and software development process. This should provide project managers with a more complete tool that can enable improved risk assessment without interfering with the work of a project's software engineers.

A second source of problems in risk management is the lack of tools [Karolak 1996]. The main reason for this lack of tools is that risk assessment is apparently an unstructured problem. To systematize unstructured problems it is necessary to define structured processes. Structured processes involve routine and repetitive problems for which a standard solution exists. Unstructured processes require decision-making based on a three-phase method (intelligence, design, choice) [Turban et al 1998]. An unstructured problem is one in which none of the three phases is structured. Current approaches to risk management are highly sensitive to managers' perceptions and preferences, which are difficult to represent by an algorithm. Depending on the decision-maker's attitude towards risk, he or she can decide early with little information, or can postpone the decision, gaining time to obtain more information, but losing some control.

A third source of risk management problems is the confusion created by the informal use of terms. Often, the software engineering community (and most parts of the project management

[2] CAPS stands for Computer Aided Prototyping System [Luqi 1988].

community [Wideman 1992]) uses the term "risk" casually. This term is often used to describe different concepts. It is erroneously used as a synonym of "uncertainty" and "threat" [SEI 1996, Hall 1997, Karolak, 1996]. Generally, software risk is viewed as a measure of the likelihood of an unsatisfactory outcome and a loss affecting the software from different points of view: project, process, and product [Hall 1997, SEI 1996]. However, this definition of risk is misleading because it confounds the concepts of risk and uncertainty. In general, most parts of decision-making in software processes are under uncertainty rather than under risk. Uncertainty is a situation in which the probability distribution for the possible outcomes is not known.

In this paper the term "risk" is reserved to indicate the probabilistic outcome of a succession of states of nature, and the term "threat" is used to identify the dangers that can occur. We define risk to be the product of the value of an outcome times its probability of occurrence. This outcome could be either positive (gain) or negative (loss). This abstraction permits one to address not only the classical risk management issue, but also to discover opportunities leading to competitive advantage.

We address the issue of risk assessment by estimating the probability distribution for the possible outcomes of a project, based on observed values of metrics that can be measured early in the process. The metrics were chosen based on a causal analysis to identify the most important threats and a statistical analysis to choose the shape of the probability distribution and relate its parameters to readily measurable metrics.

3. Related Work

There are three main groups of research related to risk:

- **Assessing Software Risk by Measuring Reliability.** This group follows a probabilistic approach and has successfully assessed the reliability of the product [Lyu 1995, Schneidewind 1975, Musa 1998]. However, this approach addresses the reliability of the product, not the risk of failing to complete the project within budget and schedule constraints. These approaches could be used to assess risks related to failures of software projects, which are outside the scope of the current paper. A concern with these approaches is that the resulting assessments arrive too late to economically correct possible faults, because the software product is mostly complete and development resources are mostly gone at the time when reliability of the product can be assessed by testing.

- **Heuristic approaches:** Other researchers assess the risk from the beginning, in parallel with the development process. However, these approaches are less rigorous, typically subjective and weakly structured. Basically these approaches use lists of practices and checklists [SEI, 1996, Hall 1997, Charette 1997, Jones 1994] or scoring techniques [Karolak 1996]. Paradoxically, SEI defines software technical risk as a measure of the probability and severity of adverse effects in developing software that does not meet its intended functions and performance requirements [SEI, 1996]. However, the term "probability" is misleading in this case because the probability distribution is unknown.

- **Macro Model Approaches:** A third group of researchers uses well known estimation models to assess how risky a project could be. The widely used methods COCOMO [Boehm 1981], and SLIM [Putnam, 1980] both assume that the requirements will remain unchanged, and require an estimation of the size of the final product as input for the models [Londeix 1987]. This size cannot be actually measured until late in the project.

The standard tools used to control all types of projects, including PERT, CPM, and Gantt, do not consider coordination and communication overhead. Such models represent sequential interdependencies through explicit representation of precedence relationships between activities. This simplified vision of a project cannot address the dynamics created by reciprocal requirements of information in concurrent activities, exception management, and the impact of actor interactions. Since the missing factors increase time requirements, the estimates resulting from these generic project estimation models are overly optimistic.

These issues are addressed by VitéProject [Levitt 1999, Thomsen et al. 1999]. VitéProject is applicable to projects in which a) all activities in the project can be predefined; b) the organization is static, and all activities are pre-assigned to actors in the static organization; c) the exceptions to activities result in extra work volume for the predefined activities and are carried out by the pre-assigned actors; and d) actors are assumed to have congruent goals. The model is well suited for simulating organizations that deal with great amounts of information processing and coordination. Such characteristics are extremely relevant in software processes [Boehm, 1981]. However, this approach requires a fixed work breakdown structure, and therefore does not apply at the early stages when requirements are changing and the set of tasks comprising the project are still uncertain.

By using informal risk assessment models, using estimation models based on optimistic assumptions that require parameters difficult to provide until late, and using optimistic project control tools, project managers condemn themselves to overrun schedules and cost.

4. The Proposed Project Risk Model

Our approach is based on metrics automatically collectable from the engineering database from near the beginning of the development. The indicators used are Requirements Volatility (RV), Complexity (CX), and Efficiency (EF).

Requirement Volatility (RV): RV is a measure of three characteristics of the requirements: a) the Birth-Rate (BR), that is the percentage of new requirements incorporated in each cycle of the evolution process; b) the Death-Rate (DR), that is the percentage of requirements dropped in each cycle; and c) the Change-Rate (CR) defined as the percentage of requirements changed from the previous version. A change in one requirement is modeled as a birth of a new requirement and the death of another, so that CR is included in the measured values of BR and DR. RV is calculated as follows: $RV = BR + DR$.

Complexity (CX): Complexity of the requirements is measured from a formal specification. A requirements representation that supports computer-aided prototyping, such as PSDL [Luqi 1996], is useful in the context of evolutionary prototyping. We define a complexity metric called Large Granularity Complexity (LGC) that is calculated as follows: $LGC = O + D + T$, where for PSDL O is the number of atomic operators (functions or state machines), D is the number of atomic data streams (data connections between operators), and T is the number of abstract data types required for the system. Operators and data streams are the components of a dataflow graph. This is a measure of the complexity of the prototype architecture, similar in spirit to function points but more suitable for modeling embedded and real-time systems. The measure can also be applied to other modeling notations that represent modules, data connections, and abstract data types or classes. We found a strong correlation between the complexity measured in LGC and the size of PSDL specifications (correlation coefficient $R = 0.996$). Most important, we also found a strong correlation ($R = 0.898$) between the complexity measured in LGC and the size of the final product expressed in non-comment lines of Ada code, including both the code automatically created by the generator and the code manually introduced by the programmers.

Efficiency (EF): The efficiency of the organization is measured using a direct observation of the use of time. EF is calculated as a ratio between the time dedicated to direct labor and the idle time: EF = Direct Labor Time / Idle Time. We found that this easily measurable quantity was a good discriminator between high team productivity and low team productivity in a set of simulated software projects [Nogueira 2000].

We validated and calibrated our model with a series of simulated software projects using VitéProject. This tool was chosen because of the inclusion of communications and exceptions in its project dynamics model, and because it has been extensively validated for many types of engineering projects, including software engineering projects. The input parameters for the simulated scenarios were RV, EF and CX, and the observed output was the development time. Given that the proposed model uses parameters collected during the early phases and given that VitéProject requires a complete breakdown structure of the project, which can be done only in the late phases, there was a considerable time gap between the two measurements. This time gap is less than for a post-mortem analysis, but it is sufficient for model calibration and validation purposes.

The simulation results were analyzed statistically, with the finding that the Weibull probability distribution was the best fit for all the samples. A random variable x is said to have a Weibull distribution with parameters α, β and γ (with $\alpha > 0$, $\beta > 0$) if the probability distribution function (pdf) and cumulative distribution function (cdf) of x are respectively:

$$\text{pdf: } f(x; \alpha, \beta, \gamma) = \begin{cases} 0, & x < \gamma \\ (\alpha/\beta^{\alpha})\,(x - \gamma)^{\alpha-1} \exp(-((x - \gamma)/\beta)^{\alpha}), & x \geq \gamma \end{cases}$$

$$\text{cdf: } F(x; \alpha, \beta, \gamma) = \begin{cases} 0, & x < \gamma \\ 1 - \exp(-((x - \gamma) / \beta)^{\alpha}) & x \geq \gamma. \end{cases}$$

The random variable under study, x, can be interpreted as development time in our context. The shape parameter α controls the skew of the pdf, which is not symmetric. We found that this is mostly related to the efficiency of the organization (EF). The scale parameter β stretches or compresses the graph in the *x* direction. We found that this parameter is related to the efficiency (EF), requirements volatility (RV), and complexity (CX) measured in LGC. The shifting parameter γ is shifts the origin of the curves to the right. We found that it is mostly related to the complexity measured in LGC.

Based on best fit to our simulation results, the model parameters can be derived from the project metrics using the following algorithm:

```
If (EF > 2.0)    then  $\alpha$ = 1.95;
                       $\gamma$ = 22 * 0.32*(13*ln(LGC)-82);
                       $\beta$ = $\gamma$ /(5.71+(RV-20)*0.046);
                 else  $\alpha$ = 2.5;
                       $\gamma$ = 22 * 0.85*(13*ln(LGC)-82);
                       $\beta$ = $\gamma$ /(5.47-(RV-20)*0.114);
                 end if;
```

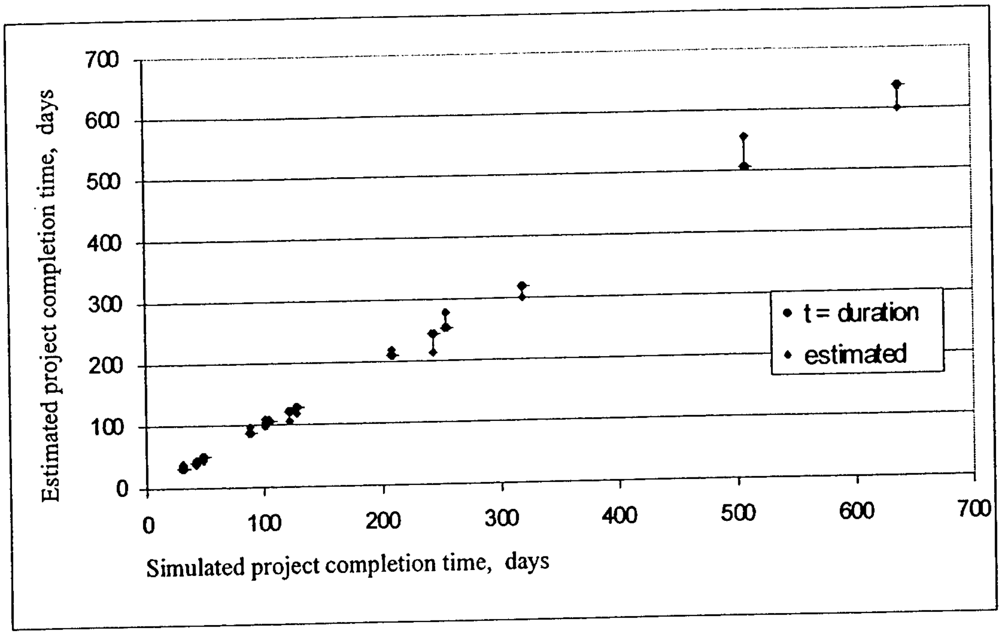
The model estimates the following cumulative probability distribution for project completion on or before time x:

```
$P(x) = 1 - \exp(-(((x - \gamma)/\beta)^{\alpha}))$ // where x is time in days
```

This equation can be inverted to obtain the schedule length needed to have a probability P of completing within schedule, with the following result.

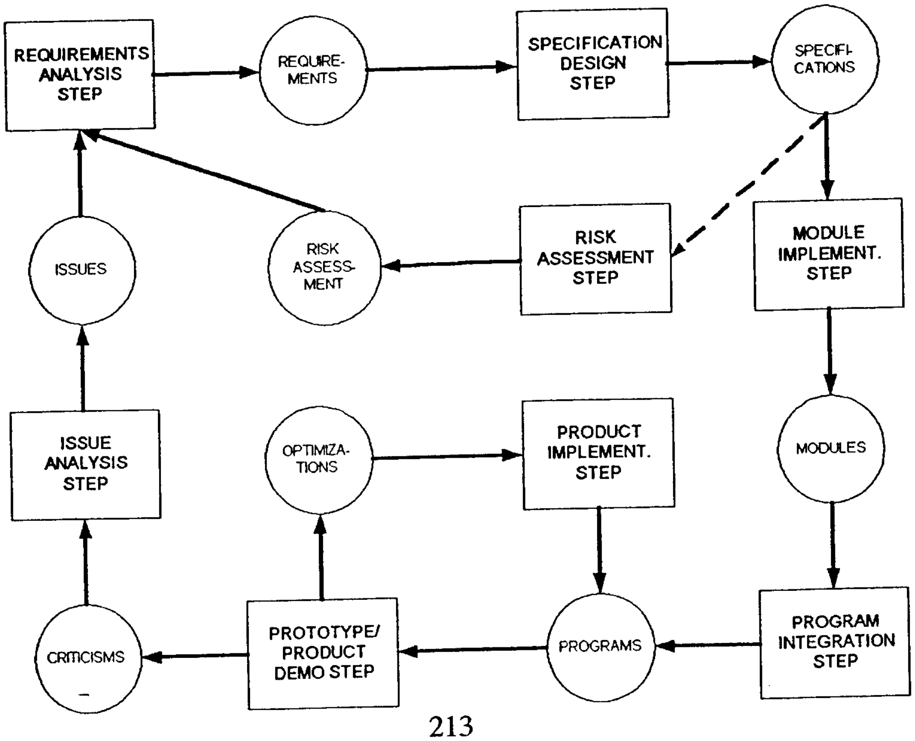
$$x = \gamma + \beta(-\ln(1-P))^{1/\alpha}$$

The probability P can be interpreted as a degree of confidence in the ability of the project to successfully complete within a schedule of length x. Applying the above equation to estimate the development time needed for a 95% chance of completion within schedule for 16 different scenarios simulated using VitéProject, we observed a standard error of 22 days. The worst case was an error of 60 days for a project of 520 days (12%). The comparison of estimated time and simulated time is shown below.



5. Integrating Risk Assessment into Prototyping

The model presented in the previous section is designed to support an iterative prototyping and software development process. In this process, an initial problem statement, a prototype demo or problem reports from a deployed software product trigger an issue analysis, followed by formulation of proposed requirements changes, and specification of a proposed adjustment to the software requirements, which can be initially empty. At this point in each cycle, the project manager should perform a risk assessment step. The results of the risk assessment step guide the degree of detail to which requirements enhancements are demonstrated, and the set of requirements issues to be considered in the next prototyping cycle, if any.



The first measurement-based risk assessment step can be performed after specification of the first version of the prototype architecture, based on the requirements volatility, LGC and efficiency measurements from the steps just performed.

In cases where risk assessments are required even earlier, before any prototyping has been done, estimates of team efficiency and requirements volatility can be based on measurements of similar past projects, and initial complexity estimates can be based on subjective guesswork of the kind currently used in the macro model approaches. This kind of estimate may be less reliable than those based solely on measurements, but it can provide a principled and reasonably accurate basis for deciding whether or not to start a prototyping process to determine the requirements for a proposed development project. Thus parts of our approach can be used truly at the very beginning of the process.

If a prototyping effort is approved, early measurements of the process could be used to refine the initial estimates of the model parameters using Bayesian methods, thus providing a balanced and systematic transition from subjective guesswork, coded as an *a priori* distribution, to assessments increasingly based on systematic measurement. Such an approach also supports incorporation and systematic refinement of measurements from previous cycles of the iterative prototyping process.

The results of risk assessment can provide guidance on the degree to which the project can afford to explore requirements enhancements requested by the customers. It can also help customers or marketing departments to decide how much they really want possible improvements, in the context of the resulting time and cost estimates. Systematic cost/benefit analysis becomes possible only with the availability of reasonably accurate estimates.

The risk assessment step can thus provide a balancing force to stabilize the requirements formulation process. In the absence of information on how much potential enhancements will cost, stakeholders are prone to unrealistic requirements amplification – of course they would always like to have a better system, no matter how good the existing one is, if you do not ask them to pay for the improvements. The proposed risk assessment steps can provide a realistic basis for incorporating time and cost constraints and cost/benefit tradeoffs early in the process, when the situation is fluid and many options are open.

This process refinement provides some additional insight into the dynamics of iterative prototyping: the iterative process should stop when the customers have determined what requirements they can afford to realize, and which of many possible improvements they will be willing to pay for, if any. It is not necessarily the case that the set of criticisms elicited by the final round of prototype demonstrations is empty – that is true only in an idealized world with adequate budgets and patient customers.

6. Conclusion

This paper introduces a formal risk assessment model for software projects based on probabilities and metrics automatically collectable from the project baseline. The approach enables a project manager to evaluate the probability of success of the project very early in the life cycle, during an iterative requirements formulation process, based on well-defined measurements rather than just guesswork or subjective judgments.

For more than twenty years, estimation standards have been characterized by a common limitation: the requirements should be frozen in order to make estimates. This model presented in this paper removes this important limitation, facing the reality that requirements are inherently variable.

The model is perfectly suited for any evolutionary software process because it follows the same philosophy. The risk assessment and estimation steps are conducted at each evolutionary cycle with increasing knowledge and decreasing variance. The research formalizes an improvement in the evolutionary software process, introducing a risk assessment step that can be automated, and that can help shape the planning of the project in the early stages when there is still substantial freedom to allocate available time and budget.

References

[Boehm 1981] B. Boehm, *Software Engineering Economics*, Prentice Hall, 1981.

[Boehm 1988] B. Boehm, A Spiral Model of Software Development and Enhancement, *Computer*, May 1988.

[Charette 1997] R. Charette, K. Adams, & M. White, Managing Risk in Software Maintenance, *IEEE Software*, May-June, 1997.

[Gilb 1977] T. Gilb, *Software Metrics*, Winthrop Publishers, Inc., 1977.

[Hall 1997] E. Hall, Managing Risk, *Methods for Software Systems Development*, Addison Wesley, 1997.

[Harn 1999] M. Harn, V. Berzins, Luqi, Computer-Aided Software Evolution Based on a Formal Model, *Proceedings of the Thirteenth International Conference on Systems Engineering*, Las Vegas, Nevada, August 9-12, 1999, pp. CS: 55-60.

[Jones 1994] C. Jones, *Assessment and Control of Software Risks*, Yourdon Press Prentice Hall, 1994.

[Karolak 1996] D. Karolak, Software *Engineering Management*, IEEE Computer Society Press, 1996.

[Levitt 1999] R. Levitt, *The ViteProject Handbook: A User's Guide to Modeling and Analyzing Project Work Processes and Organizations*, Vité © 1999.

[Londeix 1987] B. Londeix, *Cost Estimation for Software Development*, Addison-Wesley, 1987.

[Luqi 1988] Luqi, M. Ketabchi, A Computer Aided Prototyping System, *IEEE Software*, Vol. 5, No. 2, p. 66-72, March 1988.

[Luqi 1989] Luqi, Software Evolution Through Rapid Prototyping, *IEEE Computer*, May 1989.

[Luqi 1996] Luqi, Special Issue: Computer-Aided Prototyping, *Journal of Systems Integration*, Vol. 6, Nos. 1-2, March 1996.

[Lyu 1995] M. Lyu, *Software Reliability Engineering*, IEEE Computer Society Press. 1995.

[Musa 1998] J. Musa, Software *Reliability Engineering: More Reliable Software, Faster Development and Testing*, McGraw-Hill, 1998.

[Nogueira 2000] J. Nogueira, *A Formal Risk Assessment Model for Software Projects*, Ph.D. Dissertation, Naval Postgraduate School, 2000.

[Putnam 1980] L. Putnam, *Software Cost Estimating and Life-cycle Control: Getting the Software Numbers*, IEEE Computer Society Press, 1980.

[Reel 1999] J. Reel, *Critical Success Factors in Software Projects*, IEEE Software, May - June 1999.

[SEI 1996] Software Engineering Institute, Software Risk Management, Technical Report CMU/SEI-96-TR-012, June 1996.

[Schneidewind 1975] N. Schneidewind, Analysis of Error Processes in Computer Software, *Proceedings of the International Conference on Reliable Software*, IEEE Computer Society, 21-23 April 1975, p 337-346.

[Turban et al 1998] E. Turban and J. Aronson, *Decision Support Systems and Intelligent Systems*, Prentice Hall, 1998.

[vanGenuchten 1991] M. van Genuchten, Why is Software Late? An Empirical Study of the Reasons for Delay in Software Development, *IEEE Transactions on Software Engineering*, June, 1991.

[Wideman 1992] R. Wideman, *Risk Management: A Guide to Managing Project Risk Opportunities*, Project Management Institute, 1992.

Comparative Analysis of Design Alternatives in Embedded Systems*

James E. Hilger[1] Insup Lee,[2] Oleg Sokolsky[2]

[1] US Army CECOM Night Vision & Electronic Sensors Directorate

[2] Department of Computer Science, University of Pennsylvania

June 9, 2000

Abstract

The paper addresses the problem of analysis of alternatives in the design of distributed real-time systems. In the design of such a system, a hardware architecture needs to be chosen; then the system algorithm must be mapped onto the chosen architecture. As a result, the design space is very large, and evaluation of alternatives is very expensive. We propose an approach to an approximate analysis of alternative design decisions, which allows to eliminate infeasible solutions faster.

1 Introduction

Following the advances in processing and sensor technologies, more and more powerful tasks can be implemented by embedded real-time systems. Operational requirements, especially timing constraints, for such systems become ever more stringent. Parallel and distributed processing has to be used in order to meet these requirements.

Because of this, design and implementation of embedded systems is a very challenging task. With multiple processors in a system, the number of alternative implementations of an algorithm that have to be considered by designers increases dramatically. On the one hand, a hardware architecture has to be selected along with a target platform. On the other hand, the algorithm has to be mapped onto the chosen architecture, which can be done in several ways. All of these choices can have dramatic impact on the system performance, which may be hard to estimate in advance. It would be, of course, prohibitively expensive to implement several alternatives and evaluate them later. System analysis techniques have to be used in order to estimate performance of each alternative.

The most commonly used technique currently in use is simulation [2]. Simulators capable of very detailed modeling of an algorithm mapped on a distributed system are available, and allow designers to achieve highly accurate estimates. Unfortunately, detailed simulation is a very lengthy process. It would be infeasible, timewise, to analyze all design alternatives by simulation. Complete simulation of just one alternative can take months to complete. Therefore, there is a need for an alternative analysis technique that would yield results fast, even if these results would have less precision than simulation.

This paper proposes, as an alternative to simulation, an analysis methodology based on formal methods. Formal methods, a collection of specification and analysis techniques based on a mathematical description of a system, provide a rigorous way of exploring behavior of a system specification. Numerous tools are available to help the user in this exploration.

In the proposed approach, an algorithm is partitioned into a set of tasks. Each task can be run on a single processor, competing for processing time with other tasks assigned to the same processor. The user specified the amount of processing time necessary for the completion of the task, and the interconnection between tasks - that is, the size of data elements exchanged between tasks and the fre-

*This research was supported in part by NSF CCR-9619910, ARO DAAG55-98-1-0393, ARO DAAG55-98-1-0466

quency of exchanges. Data is exchanged between processors along communication channels. Several tasks may have to share a channel.

Based on this description, a formal specification of the system is constructed and analyzed by a formal methods tool. The specification is formed by a number of building blocks, each representing a system component: a processor running a certain task or set of tasks, a communication channel shared by a set of tasks, a queue for temporary storage of data, etc. Each of these components is represented by a parameterized specification, which can be instantiated according to the description of the algorithm. Since we are interested in an approximation of the system behavior, the specifications do not have to be detailed and their analysis can be performed fast, compared to simulation.

As a case study of the possibilities of this methodology, we performed analysis of an automatic target recognition (ATR) system for an Army ground vehicle. The tool used for analysis is PARAGON [3], a toolset for formal specification and analysis of real-time systems.

2 The Problem

Many mappings of an algorithm onto a multiprocessor system are usually possible. Simulation is traditionally used to evaluate different mappings and choose those that meet timing requirements for the system. However, simulation of a multiprocessor system is a laborious task, and the number of alternatives precludes us from evaluating all of them within the design cycle. Performance differences between implementations resulting from different mappings can be very significant. It is therefore important to identify those mappings that may yield feasible solutions early in the design cycle.

We approach this challenging problem by means of high-level modeling of the embedded system. Models of the system implementations are constructed and analyzed for their parameters, such as the number of processors necessary to meet the timing requirements. This analysis yields very approximate results, which nevertheless allow us to identify and discard infeasible solutions early in the design cycle.

The algorithm of the embedded system is modeled as a collection of tasks, each task assigned to a certain processor. A task corresponds to a step of the algorithm applied to a fraction of the input data. A task interacts with other tasks by sending results of its computation to the processor responsible for the subsequent round. Interaction between tasks is performed by communication channels on the processor board, or by inter-board channels. These channels are modeled to reflect their latency and transmission times. In addition, data transmitted between tasks has to be buffered to allow each task to proceed at its own speed.

The specification of an embedded system is constructed from a set of building blocks that represent commonly occuring system components. When modeled in this way, an embedded system is represented as a parallel composition of a number of similar processes. We were able to categorize the several kinds of processes involved in this kind of a specification. As mentioned above, commonly used kinds of processes are (1) computation tasks assigned to a processor, that consume inputs from one channel and produce outputs on another channel; (2) communication channels that transmit data between processors; (3) shared channels that can carry data from multiple sources to multiple destinations; (4) queues or buffers for temporary storage of data.

For each of these types of data, we defined a parameterized process template that can be easily reused in many specifications by modifying its parameters. Parameters of a template depend on the kind of a process it represents. For example, computation tasks are parameterized by the duration of the task and the processor resource that runs the task; communication channels are parameterized by their latency and bandwidth; buffers have their capacity as parameters.

This categorization of the specification components allowed us to try different system configurations quickly. Indeed, to construct a specification of a new system configuration, we just need to instantiate appropriate process templates and describe their interconnections. The process templates are organized in a library, and an intuitive user interface for instantiating and connecting templates can be provided. This turns a general-purpose formal analysis tool into a framework for analysis of embedded system designs.

3 Application

3.1 System description

We tested the described methodology in a case study involving an implementation of a specific embedded system. The algorithm used in this study is an Army Research Laboratory (ARL) sponsored ATR algorithm named the ATR Relational Template Matching (ARTM) algorithm [1]. At a high level, the ARTM algorithm operates on input images looking for edges and boundaries which, when combined, have the shapes of targets. Initial target templates are used to look for gross shapes to separate target-like regions from clutter regions. As the algorithm progresses, the target templates used are refined to separate target classes and eventually individual targets. The ARTM algorithm consists of six stages, referred to as Rounds. Rounds 0 and 1 are for screening potential target pixels from background clutter pixels (target screening). Rounds 2 and 3 perform operations on pixels that pass Rounds 0 and 1. Rounds 2 and 3 are designed to converge on a specific target pose (target separation) while Round 4 verifies the target pose (target verification). The last Round performs proximity completion, i.e. a nonlinear form of spatial integration, and outputs a final target designation list for the input image. Algorithm control flow from Round 0 to Round 4 proceeds irregularly along a conditionally branching target hypothesis tree for each pixel in the input image. Control passes to the next Round only if the latest Rounds' results exceed the threshold, else it terminates. If control terminates, the pixel under test is no longer considered as a potential target pixel. Proper threshold design permits each Round to operate on progressively fewer pixels. The image pixels included in all of these computations vary in location depending on the particular target template. The total number of operations necessary to process a single 1315 pixel by 480 pixel image according to the ARTM algorithm is approximately 8,500 million. It should be noted that this operation count includes each addition necessary to address a particular pixel location as well as operations necessary to perform mathematical computations.

The operational scenario sets overall requirements such as frequency of results, which input data, and the ultimate format of the output data. The primary function of the ATR in this case is to cue the human operator as to the location and presence of potential targets. This aids the human operator in the fatiguing task of manual search and detection. This is also necessary to meet the short timelines of target detection. It is the responsibility of the human operator to identify and prioritize targets among multiple target cues. For this study, the input data are digital InfraRed (IR) images that have 480 rows of 1315 pixels with each pixel represented by 12 bits. The input frame rate is 10 frames per second. This translates into a latency of 100 ms which represents the total time allotted to process the frame according to the algorithm.

In order to satisfy the stringent timing requirements, an implementation of the ATR system must be distributed across several processing components. The target multiprocessor system of this case study is based on 6U VME boards by Mercury Computing Systems, each containing four 400MHz PowerPC 750 processors. All integer instructions are taking 1 clock cycle, except multiplication (5 clock cycles) and division (19 cycles). Memory writes take 2 cycles. For this series of experiments, it is assumed that the superscalar integer unit of the processor allows it to execute, on the average, 1.5 integer instructions per clock cycle. The impact of cache misses degrades performance by 1.5. The bandwidth of the communication channels on the board is 160 Mbytes/s.

3.2 Analysis

The specification of the ATR system was constructed using the specification and verification toolset PARAGON [3]. PARAGON provides the means to write formal specifications and explore their behaviors. Analysis of the ATR algorithm was undertaken for several system configurations, featuring one, two and three processor boards. A sample two-board configuration is shown in Figure 1. One processor is responsible for distributing pixel data to processors on both boards. Five processors (three on board 1 and two on board 2) handle Round 0, one processor is dedicated to Round1, and all remaining rounds are processed on the last processor. While transmissions of data between processors on the same board can be carried out concurrently, inter-board transmissions all go through the same channel and are serialized. For simplicity, all transmissions between the boards are assumed to take the same amount of time. described below.

The same hardware architecture has been analyzed with a different mapping of the tasks to processors: six processors were assigned to Round 0, while Round 1 was bundled on the same processor with all other rounds.

3.3 Analysis results

Each configuration has been applied to images of different sizes and analyzed to determine the frame rate that can be achieved by the configuration. Results of the experiments are summarized in Figure 2, showing the frame rate as a function of the number of processors dedicated to Round 0.

4 Conclusions and Future Work

We presented an approach to a rapid approximate analysis of alternatives in the design of safety-critical real-time systems. The goal of analysis is to obtain estimates of the computational resources needed to implement the system, and to eliminate infeasible alternatives. Alternatives that survive this approximate analysis are then evaluated in more detail using more precise - and more expensive - techniques. Preliminary results obtained through the case study suggest that this approach will be helpful in design of large safety-critical embedded systems.

Our future research on this topic will pursue several directions in order to make this approach more helpful and easy to use.

- An intuitive front-end to the analysis tool will be designed, to shield the user as much as possible from the low-level details of the specification and analysis techniques.

- The set of process templates will be enlarged and refined. The goal of this is twofold: on the one hand, new types of templates are needed to enlarge the scope of this technique; on the other hand, templates need to be made more flexible to allow different degrees of precision during analysis.

- Additional case studies need to be performed to gain more experience, which will allow us to improve the technique further.

References

[1] T. Kipp *et al.* ATR Relational Template Matching, Phase 1 Final S & T, Report, Army Research Laboratory, Alliant Techsystems, Inc., and Mathematical Technologies, Inc. Contract DAAB07-90-C-F427, 1990.

[2] J. E. Hilger. Adaptive computing technology: An enabling processing technology for advanced sensor systems. In *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, October 1998.

[3] O. Sokolsky, I. Lee, and H. Ben-Abdallah. Specification and analysis of real-time systems with PARAGON. *Annals of Software Engineering*, 1999. Accepted for publication.

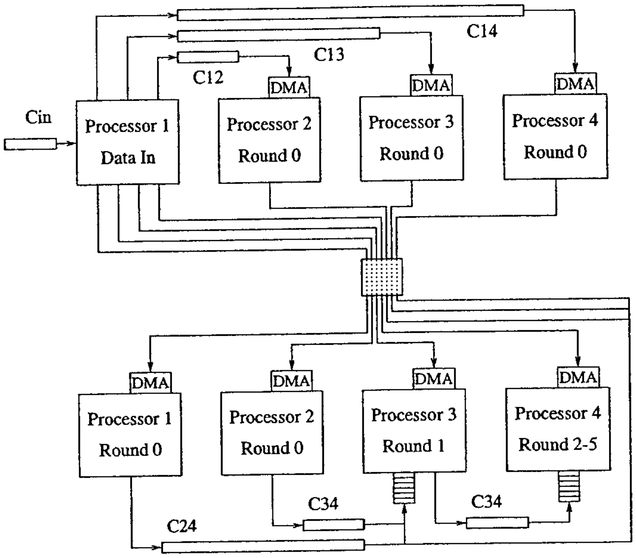


Figure 1: Sample ATR architecture

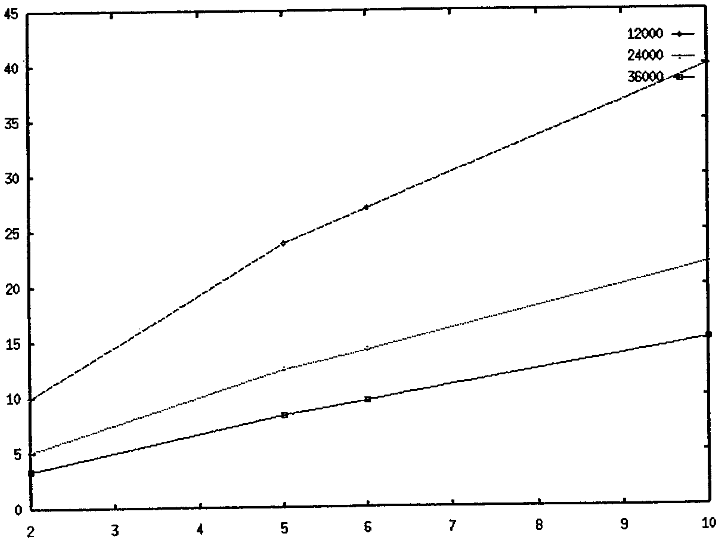


Figure 2: Frame rate as function of Round 0 processors

Dependability of Computer-Based Systems

Cliff B Jones

Department of Computing Science,
University of Newcastle
NE1 7RU, UK
e-mail: cliff.jones@ncl.ac.uk

Abstract. This paper sets out a programme of work in the area of dependability. The research is to be pursued under the aegis of a six-year *Inter-Disciplinary Research Collaboration* funded by the UK Engineering and Physical Sciences Research Council. The research considers computer-based systems which comprise humans as well as hardware and software. The aim here is to indicate how formal methods ideas, coupled with structuring proposals, can help address a problem which clearly also requires social science input.

EXTENDED ABSTRACT

Reasoning about interference

This section summarises earlier work on formal development methods for concurrent systems.

The essence of concurrency is interference: shared-variable programs must be designed so as to tolerate state changes; communication-based concurrency shifts the interference to that from messages. One possible way of specifying interference is to use rely/guarantee-conditions (see [Jon83, Sti88, Stø90, Xu92, Col94, Din99]).

Programming language designers have proposed a series of increasingly sophisticated constructs to control interference; the case for using object-oriented constructs is set out in [Jon93].

Faults as interference

The essence of this section is to argue that faults can be viewed as interference in the same way that concurrent processes bring about changes beyond the control of the process whose specification and design are being considered. Without yet proposing notation for each case, a range of motivating examples are considered.

The first example is one that re-awakened this author's interest in considering faults as interference. Faced with the task of specifying a traffic light system, many computer scientists would confine themselves to the control system and specify the signals which must be emitted. Michael Jackson (see [Jac00]) considers the wider issues of the correct wiring of the control system to the physical lights and the initial state of these lights units. One could widen the specification to address the overall light system (at one level the requirement is that at least one light must always be red) and record assumptions (as rely-conditions) which state that emitting a signal implies that the light unit changes state. Recording such a rely-condition does not itself result in a dependable system but it ensures that the assumptions are recorded and use of proof rules for development of concurrency should ensure that there are no further hidden assumptions. In fact, one could take this example further by specifying that the real requirement is to reduce the probability of a crash to a certain level and then record probabilities that drivers behave in certain ways when faced with red lights (see, for example, [MMS96] for ways of reasoning about probabilities in design).

A second –trivial– example should again illustrate the shift of view in documenting assumptions. Rather than specifying a control system in terms of the readings delivered by measuring devices, it might be preferable to specify the overall system in terms of the actual temperature etc. and provide a rely-condition which records the acceptable tolerance on the measuring device. Here again, the message is to expose the assumptions.

A more realistic example can be given in the same domain: it would be common for such sensors to be deployed using "triple modular redundancy". The viewpoint of recording the assumptions would suggest that a rely-condition should state that two roughly equal measurements are far less likely to be in error than one which is wildly different (or is perhaps some distinguished error value).

As well as the primary message that exposing assumptions will force their consideration, there is the clear advantage that checking that such rely-conditions are adequate to prove that a system will meet its overall specification will check for any missed assumptions.

Fault containment and recovery

Significant work has been done on designing architectures for fault-containment and recovery – see for example [Ran75, XRR$^+$99].

Human errors and their containment

The work in the *Dependability Interdisciplinary Research Collaboration* on which we are embarking will address not just dependable computer systems but will also consider wider systems where the role of the humans involved is seen as critical to overall system dependability. The need for this is emphasized by [Mac94] which reports a large number of computer related accidents which resulted in death and notes that in the majority of cases the key problem related more to the interaction between people and computers than a specific hardware or software malfunction.

There are of course many examples of where a program tries to guard against inadvertent errors of its users: the check in many operating systems asking a user to confirm the request to delete files or the need to retype a new password (being invisible there is no other check) are trivial instances. More interesting is the architecture of the overall system known as Pen& Pad [HRH+90] in which software is programmed to warn against possible misprescription of drugs by doctors: no attempt was made to automate prescription but the system would check against dangerous cocktails or specific drugs which might not be tolerable to some other condition that is indicated on the patient's record.

The logical extension of the work outlined in the two preceding sections on purely computer systems is to aim for a more systematic treatment of human errors. Fortunately the work of psychologists like Reason (see [Rea90]) in categorising human errors offers the hope of describing and reasoning about the sort of human errors against which a system is designed to guard. The objective would be to minimise the risk of the errors of the computer system and (groups of) humans "lining up" in the way indicated in [Rea97].

Further research

There are many further areas of research related to the themes above. For example:

- Both pre and rely-conditions can record assumptions but if they become complex they might be a warning that an interface has become too messy (cf. [CJ00]) – ways of evaluating interfaces and architectures are needed (see [SG96]).
- The idea of using rely-conditions to record failure assumptions occurred to the author in a connection with a control system some years ago. One reason for not describing the idea more publicly was that there often appears to be a mismatch of abstraction levels between the specification and the error inducing level. There needs to be more research on whether this can be avoided.
- The role of malicious attacks is being considered in the IST-funded MAFTIA project.
- A key area of system "misuse" is where the user has an incorrect model of what is going on inside the combined control/controlled system – minimizing this risk must be an objective.
- Progress in modelling the human mind (e.g. [CS94]) should be tracked.

Acknowledgements

There is of course related research and the work of Michael Harrison and his colleagues (who are within the IRC) and John Rushby (see [Rus99]) has influenced the thinking so far. One particular stimulus for this paper was the talk that Michael Jackson gave at the Munich meeting of IFIP's WG2.3 last year – more generally discussions at this working group on Programming Methodology have acted as a sounding board and encouragement to the author. This extended

abstract was published earlier in the proceedings of MPC'2000. Evolving details of the Dependability IRC can be found at www.dirc.org.uk.

References

[CJ00] Pierre Collette and Cliff B. Jones. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In G. D. Plotkin, editor, *Proof, Language and Interaction*, chapter 10, pages 275–305. MIT Press, 2000.

[Col94] Pierre Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications – Application to UNITY.* PhD thesis, Louvain-la-Neuve, June 1994.

[CS94] Patricia S Churchland and Terrance J Sejnowski. *The Computational Brain.* MIT Press, 1994.

[Din99] Jürgen Dingel. *Systematic Parallel Programming.* PhD thesis, Carnegie Mellon University, 1999.

[HRH$^+$90] T J Howkins, A L Rector, C A Horan, A Nowlan, and A Wilson. An overview of PEN& PAD. *Lecture Notes in Medical Informatics*, 40:73–78, 1990.

[Jac00] Michael Jackson. *Problem Frames: Structring and Analysing Software Development Problems.* Addison-Wesley, 2000.

[Jon83] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[Jon93] C. B. Jones. Constraining interference in an object-based design method. In M-C. Gaudel and J-P. Jouannaud, editors, *TAPSOFT'93*, volume 668 of *Lecture Notes in Computer Science*, pages 136–150. Springer-Verlag, 1993.

[Mac94] Donald MacKenzie. Computer-related accidental death: an empirical exploration. *Science and Public Policy*, 21:233–248, 1994.

[MMS96] Carroll Morgan, Annabelle McIver, and J W Sanders. Refinement-oriented probability for CSP. *Formal Aspects of Computing*, 8(6):617–647, 1996.

[Ran75] B. Randell. System structure for fault tolerance. *IEEE Transactionns on Software Engineering*, SE-1:220–232, 1975.

[Rea90] James Reason. *Human Error.* Cambridge University Press, 1990.

[Rea97] James Reason. *Managing the Risks of Organisational Accidents.* Ashgate Publishing Limited, 1997.

[Rus99] John Rushby. Using model checking to help discover mode confusions and other automation surprises. In *Proceedings of 3rd Workshop on Human Error*, pages 1–18. HESSD'99, 1999.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[Sti88] C. Stirling. A generalisation of Owicki-Gries's Hoare logic for a concurrent while language. *TCS*, 58:347–359, 1988.

[Stø90] K. Stølen. *Development of Parallel Programs on Shared Data-Structures.* PhD thesis, Manchester University, 1990. available as UMCS-91-1-1.

[XRR$^+$99] J. Xu, B. Randell, A. Romanovsky, R. J. Stroud, A. F. Zorzo, E. Canver, and F. von Henke. Rigorous development os a safety-critical system based on coordinated atomic actions. In *Proc. of 29th Int. Symp. Fault-Tollerant Computing.* IEEE Computer Society Press, 1999.

[Xu92] Qiwen Xu. *A Theory of State-based Parallel Programming.* PhD thesis, Oxford University, 1992.

This article was processed using the LaTeX macro package with LLNCS style

Verification Diagrams: Logic + Automata

Zohar Manna and Henny B. Sipma *

Computer Science Department
Stanford University
Stanford, CA. 94305-9045
{manna,sipma}@cs.stanford.edu

Abstract. We use automata on infinite words to reduce the verification of linear temporal logic (LTL) properties over infinite-state systems to the proof of first-order verification conditions and an algorithmic language inclusion check. The automaton serves as a temporal abstraction of the system, preserving a subset of both safety and liveness properties. The first-order verification conditions prove that the abstraction is conservative; the algorithmic check verifies that the abstraction satisfies the property. Automata precisely separate the combinatoric from the logic part of the proof, such that the combinatoric part can be handled completely by algorithmic methods.

1 Introduction

Verification diagrams cleanly separate combinatorics, handled by the underlying automata, from logic, represented by first-order verification conditions, in the proof that a reactive system satisfies a temporal specification. Automata are ubiquitous in program verification. However, all of their use has been in model checking [Kur94,VW86], the combinatoric part of the proof: both the system and the negation of the property are represented as a finite-state automaton and property satisfaction is checked by means of a decidable emptiness check of the product automaton. In this paper we show that automata can also successfully be used in the verification of infinite-state systems in the form of ***verification diagrams*** [MP94]. These diagrams are temporal abstractions of the system that preserve liveness properties: the acceptance condition of the automaton restricts the infinite behavior of the abstract system [BMS95,MBSU98].

To show that a system $\mathcal{S}$ satisfies a temporal property φ, a verification diagram $\mathcal{G}$ is constructed such that the language inclusion (where the language of a diagram is similar to that of the underlying automaton)

$$\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\mathcal{G})$$

* This research was supported in part by the National Science Foundation under grant CCR-98-04100 and CCR-99-00984 ARO under grants DAAH04-96-1-0122 and DAAG55-98-1-0471, ARO under MURI grant DAAH04-96-1-0341, by Army contract DABT63-96-C-0096 (DARPA), and by Air Force contract F33615-99-C-3014.

can be proved by first-order verification conditions, and the language inclusion

$$\mathcal{L}(\mathcal{G}) \subseteq \mathcal{L}(\varphi)$$

can be proved algorithmically, thus separating the deductive and algorithmic parts of the proof, and eliminating the need to perform any deductive temporal reasoning.

Construction of the diagram may be an iterative process, starting with the diagram based on the automaton for the property and refining this diagram until all first-order verification conditions can be proved. In this case the diagram is guaranteed to satisfy the property. The verification diagram is a true abstraction of the system in the same domain: it over approximates the set of computations of the system.

Like in model checking one can also start with a diagram based on the automaton for the negation of the property. The resulting *falsification diagram* [SUM99] over approximates the set of computations of the system that do not satisfy the property. The goal is now to refine the diagram, justified by first-order verification conditions, until it is empty, proving that no computation satisfies the negation of the property. This process is called *Deductive Model Checking*.

Both verification diagrams and falsification diagrams take as starting point a nondeterministic ω-automaton [Tho88] for the (negation of the) property. The size of the automaton is worst-case exponential in the size of the property, which is undesirable, since the number of first-order verification conditions is proportional to the size of the automaton. Recently we have investigated *alternating automata*, which are linear in the size of the property, as the basis for diagrams and verification rules [MS00].

In this paper we will give an overview of the use of diagrams in verification. The remainder of the paper is organized as follows. Section 2 provides the preliminaries: our computational model of fair transition systems, our specification language of linear temporal logic (LTL), and the basics of ω-automata. Section 3 presents verification diagrams, separated in the logic part and the combinatoric part. Sections 4, 5 and 6 introduce alternating automata, and show how they can be used to reduce the proof of an LTL property to a set of first-order verification conditions.

2 Preliminaries

2.1 Computational Model: Fair Transition Systems

The computational model used for reactive systems is that of a *transition system* [MP95] (TS), $\mathcal{S} = \langle V, \Theta_{\mathcal{S}}, \mathcal{T} \rangle$, where V is a finite set of variables, $\Theta_{\mathcal{S}}$ is an initial condition, and $\mathcal{T}$ is a finite set of transitions. A *state* s is an interpretation of V, and Σ denotes the set of all states. A transition $\tau \in \mathcal{T}$ is a function $\tau: \Sigma \mapsto 2^{\Sigma}$, and each state in $\tau(s)$ is called a τ-successor of s. We say that a transition τ is *enabled* on s if $\tau(s) \neq \emptyset$, otherwise τ is *disabled* on s. Each transition τ is represented by a *transition relation* $\rho_\tau(s, s')$, an assertion that expresses the relation between the values of V in s and the values of V (referred to by V') in any of its τ-successors s'.

A *run* of $\mathcal{S}$ is an infinite sequence of states such that the first state satisfies $\Theta_{\mathcal{S}}$ and any two consecutive states satisfy a ρ_τ for some $\tau \in \mathcal{T}$. A state s is called *$\mathcal{S}$-accessible* if it appears in some run of $\mathcal{S}$.

Transitions can be marked as *just* or *compassionate*. Just (or *weakly fair*) transitions cannot be continuously enabled without ever being taken. Compassionate (or *strongly fair*) transitions cannot be enabled infinitely often without being taken. Every compassionate transition is also just. A *computation* is a run that satisfies these fairness requirements. The set of all computations of $\mathcal{S}$ is denoted by $\mathcal{L}(\mathcal{S})$.

2.2 Specification Language: Linear Temporal Logic

The specification language studied in this paper is *linear temporal logic*. We assume an underlying *assertion language* which is a first-order language over interpreted symbols for expressing functions and relations over some concrete domains. We refer to a formula in the assertion language as a *state formula* or *assertion*. A *temporal formula* is constructed out of state formulas to which we apply the boolean connectives and the temporal operators shown below.

Temporal formulas are interpreted over a *model*, which is an infinite sequence of states $\sigma : s_0, s_1, \ldots$. Given a model σ, a state formula p and temporal formulas φ and ψ, we present an inductive definition for the notion of a formula φ holding at a position $j \geq 0$ in σ, denoted by $(\sigma, j) \models \varphi$.

For a state formula:

$$(\sigma, j) \models p \quad \text{iff} \quad s_j \models p, \quad \text{that is, } p \text{ holds on state } s_j.$$

For the boolean connectives:

$$\begin{array}{lll} (\sigma,j) \models \phi \wedge \psi & \text{iff} & (\sigma,j) \models \phi \text{ and } (\sigma,j) \models \psi \\ (\sigma,j) \models \phi \vee \psi & \text{iff} & (\sigma,j) \models \phi \text{ or } (\sigma,j) \models \psi \\ (\sigma,j) \models \neg\phi & \text{iff} & (\sigma,j) \not\models \phi \ . \end{array}$$

For the future temporal operators:

$$\begin{array}{lll} (\sigma,j) \models \bigcirc \phi & \text{iff} & (\sigma,j+1) \models \phi \\ (\sigma,j) \models \Box \phi & \text{iff} & (\sigma,i) \models \phi \text{ for all } i \geq j \\ (\sigma,j) \models \Diamond \phi & \text{iff} & (\sigma,i) \models \phi \text{ for some } i \geq j \\ (\sigma,j) \models \phi \,\mathcal{U}\, \psi & \text{iff} & (\sigma,k) \models \psi \text{ for some } k \geq j, \\ & & \quad \text{and } (\sigma,i) \models \phi \text{ for every } i,\ j \leq i < k \\ (\sigma,j) \models \phi \,\mathcal{W}\, \psi & \text{iff} & (\sigma,j) \models \phi \,\mathcal{U}\, \psi \text{ or } (\sigma,j) \models \Box \phi \ . \end{array}$$

For simplicity of the presentation, we will omit the past temporal operators in this paper. However both verification diagrams and the alternating automata are applicable to LTL formulas that include past operators. An infinite sequence of states σ *satisfies* a temporal formula ϕ, written $\sigma \models \phi$, if $(\sigma, 0) \models \phi$. The set of all sequences that satisfy a formula φ is denoted by $\mathcal{L}(\varphi)$, the *language* of φ.

We say that a formula is a future formula if it contains only state formulas, boolean connectives and future temporal operators. We say that a formula is a general safety formula if it is of the form $\Box\varphi$, for a past formula φ.

A state formula p is called $\mathcal{S}$-*state valid* if it holds over all $\mathcal{S}$-accessible states. A temporal formula φ is called $\mathcal{S}$-*valid* (valid over system $\mathcal{S}$), denoted by

$$\mathcal{S} \vDash \varphi \ ,$$

if it holds over all computations of $\mathcal{S}$.

2.3 Nondeterministic ω-automata

Verification diagrams are based on nondeterministic ω-automata. Automata are represented by a tuple $\mathcal{A} : \langle N, N_0, E, \nu, \mathcal{F}\rangle$, where N (N_0) are the (initial) nodes, E are the edges, ν is the node labeling, a function from the set of nodes to boolean expressions over atomic assertions, and $\mathcal{F}$ is the acceptance condition, in our case a set of subsets of nodes, also known as a Muller acceptance condition.

An infinite sequence of nodes $\pi : n_0, n_1, \ldots$ is called a path of an automaton $\mathcal{A}$ if n_0 is an initial node, and for each $i > 0$, $\langle n_i, n_{i+1}\rangle \in E$. The set of nodes that appear infinitely often in π is called the limit set of π, written $\mathit{inf}(\pi)$. A path π is accepting if $\mathit{inf}(\pi) \in \mathcal{F}$. The set $\mathit{inf}(\pi)$ must necessarily form a *strongly connected subgraph* (SCS) in the automaton, that is each node in $\mathit{inf}(\pi)$ can be reached from every other node in $\mathit{inf}(\pi)$ without leaving this set.

A sequence of states $s_0, s_1, \ldots$ is a model of $\mathcal{A}$ if there exists an accepting path $n_0, n_1, \ldots$ in $\mathcal{A}$ such that for all $i \geq 0$ $s_i \vDash \nu(n_i)$. The set of models of $\mathcal{A}$ is called the language of $\mathcal{A}$, written $\mathcal{L}(\mathcal{A})$.

3 Verification Diagrams

Verification diagrams are a complete proof method (relative to first-order reasoning) to prove arbitrary state quantified LTL properties over infinite-state systems.

Verification diagrams [MP94,BMS95,BMS96,MBSU98] are nondeterministic ω-automata augmented with an additional node labeling μ, a function from nodes to assertions over the program variables. A sequence of states $s_0, s_1, \ldots$ is a model of a verification diagram if there exists an accepting path $\pi : n_0, n_1, \ldots$ in the diagram (that is, accepted by the underlying automaton) such that for every $i \geq 0$, $s_i \vDash \mu(n_i)$. The language of the diagram $\mathcal{G}$, written $\mathcal{L}(\mathcal{G})$, is the set of all its models. The underlying automaton of a diagram $\mathcal{G}$ is denoted by $\mathcal{G}_{\mathcal{A}}$.

3.1 Verification Diagrams: The Logic Part

To show that all computations of a system $\mathcal{S}$ are included in the language of a diagram $\mathcal{G}$, that is, $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\mathcal{G})$, we have to show

Initiation Every initial state of S must be able to be mapped onto some initial state of the diagram. This holds if the following condition holds:

$$\Theta \rightarrow \mu(N_0)$$

where $\mu(S)$ with $S = \{n_1, \ldots, n_k\} \subseteq N$ stands for

$$\mu(S) \overset{\text{def}}{=} \mu(n_1) \vee \ldots \vee \mu(n_k)$$

It states that every run of S can start at some initial node of $\mathcal{G}$.

Consecution For every node $n \in N$ and for every state $s \models \mu(n)$, every successor state of s must be able to be mapped onto a successor node of n. This holds if the following condition holds for every node $n \in N$:

$$\mu(n) \wedge \rho_\tau \rightarrow \mu'(succ(n))$$

where $succ(n)$ stands for all successor nodes of n.

Acceptance The acceptance condition of the automaton eliminates from the diagram language all sequences of states all of whose paths end up in a nonaccepting SCS. We have to show that none of these sequences correspond to a computation of the system. We say that an SCS S is *transient* if every computation of S with a path ending in S has a way of leaving S. To show that every computation has at least one accepting path in the diagram it suffices to show that every nonaccepting SCS is transient[Sip99]. An SCS can be shown to be transient in one of the following three ways:

Just exit An SCS S has a just exit, if there is a just transition τ such that the following verification conditions hold for every node $m \in S$:

$$\mu(m) \rightarrow enabled(\tau)$$

and

$$\mu(m) \wedge \rho_\tau \rightarrow \mu'(succ(m) - S)$$

The first condition states that τ is enabled on every node, and the second condition ensures that the computation can leave the SCS at every node.

Compassionate exit An SCS has a compassionate exit, if there is a compassionate transition τ such that the following conditions hold for every node $m \in S$:

$$\mu(m) \rightarrow \neg enabled(\tau)$$

or

$$\mu(m) \wedge \rho_\tau \rightarrow \mu'(succ(m) - S)$$

and for some node $n \in S$, τ is enabled at n:

$$\mu(n) \rightarrow enabled(\tau)$$

This states that for eveyr node in S either τ is disabled or τ can lead out of S, and there is at least one node n where τ can indeed leave S.

Well-founded SCS An SCS $S : \{n_1, \ldots, n_k\}$ is *well-founded* if there exist ranking functions $\{\delta_1, \ldots, \delta_k\}$, where each δ_i maps the system states into elements of a well-founded domain $(\mathcal{D}, \succ)$, such that the following verification conditions are valid: there is a *cut-set*[1] E of edges in S such that for all edges $(n_1, n_2) \in E$ and every transition τ,

$$\mu(n_1) \wedge \rho_\tau \wedge \mu'(n_2) \rightarrow \delta_1 \succ \delta_2' ,$$

and for all other edges $(n_1, n_2) \notin E$ in S and for all transitions τ,

$$\mu(n_1) \wedge \rho_\tau \wedge \mu'(n_2) \rightarrow \delta_1 \succeq \delta_2' .$$

This means that there is no computation that ends up in S: it would have to traverse at least one of the edges in E infinitely often, which contradicts the well-foundedness of the ranking functions.

In addition, we need to show that the language of the diagram is included in the language of the underlying automaton of the diagram, that is

$$\mathcal{L}(\mathcal{G}) \subseteq \mathcal{L}(\mathcal{G}_A)$$

This holds if the following first-order verification holds for every node n in $\mathcal{G}$:

$$\mu(n) \rightarrow \nu(n) .$$

Thus, if the above verification conditions hold it is ensured that every computation of the system is represented in the language of the underlying automaton of the diagram.

3.2 Verification Diagrams: The Automata Part

Having shown $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\mathcal{G}_A)$ it remains to show that all models of the underlying automaton of the diagram satisfy the property, that is

$$\mathcal{L}(\mathcal{G}_A) \subseteq \mathcal{L}(\varphi) .$$

This check can be performed using a straightforward abstraction and standard ω-automata model checking.

Let $B = \{b_1, \ldots, b_n\}$ be the set of first-order atomic formulas appearing in the property φ to be proven. Abstracting both the automaton for the negation of the property and the underlying automaton of the diagram with the abstraction function that in each node labeling replaces each atomic formula with the corresponding proposition of the boolean algebra over B, we obtain two finite-state ω-automata, $\mathcal{G}_A^\alpha$ and $\mathcal{A}^\alpha(\neg\varphi)$, and we can check the language inclusion $\mathcal{L}(\mathcal{G}_A^\alpha) \subseteq \mathcal{L}(\mathcal{A}^\alpha(\varphi))$ by checking $\mathcal{L}(\mathcal{G}_A^\alpha \times \mathcal{A}^\alpha(\neg\varphi))$ for emptiness.

[1] A cut-set of an SCS S is a set of edges E such that every loop in S contains some edge in E (that is, the removal of E disconnects S).

It is easy to show that the abstraction function and its corresponding concretization function (which replaces in each node labeling every proposition with the corresponding assertion) form a Galois insertion, and thus from

$$\mathcal{L}(\mathcal{G}_A^\alpha) \subseteq \mathcal{L}(\mathcal{A}^\alpha(\varphi))$$

we can conclude

$$\mathcal{L}(\mathcal{G}_A) \subseteq \mathcal{L}(\mathcal{A}(\varphi))$$

as required.

3.3 Verification Diagrams: Semi-Automatic Generation

As mentioned in the Introduction one can take the automaton for the property as a starting point for the verification diagram. The task at hand is now to refine the diagram by splitting nodes and strengthening the assertions labeling the nodes until the verification conditions associated with the diagram hold. If the diagram is refined in this manner, the combinatoric check becomes redundant, since the diagram is guaranteed to satisfy the property.

The disadvantage of this approach is that the diagram may get very large, since the size of the automaton is worst-case exponential in the size of the property. In the next section we introduce alternating automata, which are linear in the size of the property, to alleviate this problem to a certain extent.

4 Alternating Automata

Alternating automata are a generalization of nondeterministic automata. Nondeterministic automata have an existential flavor: a word is accepted if it is accepted by *some* path through the automaton. On the other hand ∀-automata [MP87] have a universal flavor: a word is accepted if it is accepted by *all* paths. Alternating automata combine the two flavors by allowing choices along a path to be marked as either existential or universal.

An *alternating automaton* $\mathcal{A}$ is defined recursively as follows:

$\mathcal{A} ::= \epsilon_{\mathcal{A}}$	empty automaton
$\mid\ \langle \nu, \delta, f \rangle$	single node
$\mid\ \mathcal{A} \wedge \mathcal{A}$	conjunction of two automata
$\mid\ \mathcal{A} \vee \mathcal{A}$	disjunction of two automata

where ν is a state formula, δ is an alternating automaton expressing the next-state relation, and f indicates whether the node is accepting (denoted by $+$) or rejecting (denoted by $-$). We require that the automaton be finite.

The set of nodes of an alternating automaton $\mathcal{A}$, denoted by $\mathcal{N}(\mathcal{A})$ is formally defined as

$$\begin{aligned}
\mathcal{N}(\epsilon_{\mathcal{A}}) &= \emptyset \\
\mathcal{N}(\langle \nu, \delta, f \rangle) &= \langle \nu, \delta, f \rangle \cup \mathcal{N}(\delta) \\
\mathcal{N}(\mathcal{A}_1 \wedge \mathcal{A}_2) &= \mathcal{N}(\mathcal{A}_1) \cup \mathcal{N}(\mathcal{A}_2) \\
\mathcal{N}(\mathcal{A}_1 \vee \mathcal{A}_2) &= \mathcal{N}(\mathcal{A}_1) \cup \mathcal{N}(\mathcal{A}_2)
\end{aligned}$$

A path through a regular ω-automaton is an infinite sequence of nodes. A "path" through an alternating ω-automaton is, in general, a tree. A *tree* is defined recursively as follows:

$$\begin{array}{lll} T ::= & \epsilon_T & \text{empty tree} \\ \mid & T \cdot T & \text{composition} \\ \mid & \langle node, T \rangle & \text{single node with child tree} \end{array}$$

A tree may have both finite and infinite branches.

Given an infinite sequence of states $\sigma : s_0, s_1, \ldots$, a tree T is called a ***run*** of σ in $\mathcal{A}$ if one of the following holds:

$$\begin{array}{lll} \mathcal{A} = \epsilon_{\mathcal{A}} & \text{and} & T = \epsilon_T \\ \mathcal{A} = n & \text{and} & T = \langle n, T' \rangle \text{ and } s_0 \vDash \nu(n) \text{ and} \\ & & T' \text{ is a run of } s_1, s_2, \ldots \text{ in } \delta(n) \\ \mathcal{A} = \mathcal{A}_1 \wedge \mathcal{A}_2 & \text{and} & T = T_1 \cdot T_2, \\ & & T_1 \text{ is a run of } \mathcal{A}_1 \text{ and } T_2 \text{ is a run of } \mathcal{A}_2 \\ \mathcal{A} = \mathcal{A}_1 \vee \mathcal{A}_2 & \text{and} & T \text{ is a run of } \mathcal{A}_1 \text{ or } T \text{ is a run of } \mathcal{A}_2 \end{array}$$

A run T is ***accepting*** if every infinite branch contains infinitely many accepting nodes. An infinite sequence of states σ is a ***model*** of an alternating automaton $\mathcal{A}$ if there exists an accepting run of σ in $\mathcal{A}$. The set of models of an automaton $\mathcal{A}$, also called the ***language of*** $\mathcal{A}$, is denoted by $\mathcal{L}(\mathcal{A})$.

5 Translating LTL formulas into Alternating Automata

It has been shown that for every LTL formula φ there exists an alternating automaton $\mathcal{A}$ such that $\mathcal{L}(\varphi) = \mathcal{L}(\mathcal{A})$ and the size of $\mathcal{A}$ is linear in the size of φ [Var97]. In [Var97] a construction method is given for such an automaton with propositions labeling the edges. Since we prefer to label the nodes with propositions (or, in our case, state formulas), we present a slightly different procedure. In the remainder of this paper we assume that all negations have been pushed in to the state level (a full set of rewrite rules to accomplish this is given in [MP95]), that is, no temporal operator is in the scope of a negation.

Given an LTL formula φ, an alternating automaton $\mathcal{A}(\varphi)$ is constructed, as follows.

For a state formula p:

$$\mathcal{A}(p) = \langle p, \epsilon_{\mathcal{A}}, + \rangle \ .$$

For temporal formulas φ and ψ:

$$\begin{array}{lcl} \mathcal{A}(\varphi \wedge \psi) & = & \mathcal{A}(\varphi) \wedge \mathcal{A}(\psi) \\ \mathcal{A}(\varphi \vee \psi) & = & \mathcal{A}(\varphi) \vee \mathcal{A}(\psi) \\ \mathcal{A}(\bigcirc \varphi) & = & \langle true, \mathcal{A}(\varphi), + \rangle \\ \mathcal{A}(\Box \varphi) & = & \langle true, \mathcal{A}(\Box \varphi), + \rangle \wedge \mathcal{A}(\varphi) \\ \mathcal{A}(\Diamond \varphi) & = & \langle true, \mathcal{A}(\Diamond \varphi), - \rangle \vee \mathcal{A}(\varphi) \\ \mathcal{A}(\varphi \mathcal{U} \psi) & = & \mathcal{A}(\psi) \vee (\langle true, \mathcal{A}(\varphi \mathcal{U} \psi), - \rangle \wedge \mathcal{A}(\varphi)) \\ \mathcal{A}(\varphi \mathcal{W} \psi) & = & \mathcal{A}(\psi) \vee (\langle true, \mathcal{A}(\varphi \mathcal{W} \psi), + \rangle \wedge \mathcal{A}(\varphi)) \end{array}$$

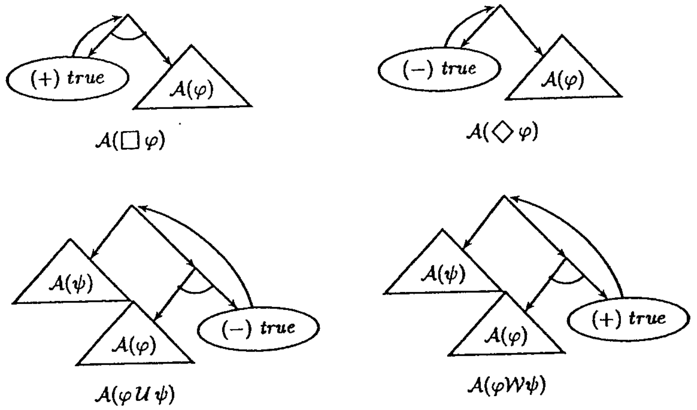


Fig. 1. Alternating automata for the temporal operators $\Box$, $\Diamond$, $\mathcal{U}$, $\mathcal{W}$

The constructions for the temporal formulas are illustrated in Figure 1.

In [MS00] it is shown that for a future temporal formula φ, $\mathcal{L}(\varphi) = \mathcal{L}(\mathcal{A}(\varphi))$.

6 Temporal Verification Rule for Future Safety Formulas

Alternating automata can be used to automatically reduce the verification of an arbitrary safety property specified by a future formula to first-order verification conditions, where a safety property is defined to be a property φ, such that if a sequence σ does not satisfy φ, then there is a finite prefix of σ such that φ is false on every extension of this prefix.

We define the ***initial condition*** of an alternating automaton $\mathcal{A}$, denoted by $\theta_\mathcal{A}(\mathcal{A})$, as follows:

$$\begin{array}{lcl} \theta_\mathcal{A}(\epsilon_\mathcal{A}) & = & \mathit{true} \\ \theta_\mathcal{A}(\langle \nu, \delta, f \rangle) & = & \nu \\ \theta_\mathcal{A}(\mathcal{A}_1 \wedge \mathcal{A}_2) & = & \theta_\mathcal{A}(\mathcal{A}_1) \wedge \theta_\mathcal{A}(\mathcal{A}_2) \\ \theta_\mathcal{A}(\mathcal{A}_1 \vee \mathcal{A}_2) & = & \theta_\mathcal{A}(\mathcal{A}_1) \vee \theta_\mathcal{A}(\mathcal{A}_2) \end{array}$$

Intuitively, the initial condition of an automaton characterizes the set of initial states of sequences accepted by the automaton.

Basic Rule

Following the style of verification rules of [MP95] we can now present the basic temporal rule B-SAFE, shown in Figure 2. In the rule we use the *Hoare triple* notation $\{p\}\,\tau\,\{q\}$, which stands for $p \wedge \rho_\tau \to q'$. The notation $\{p\}\,\mathcal{T}\,\{q\}$ stands for $\{p\}\,\tau\,\{q\}$ for all $\tau \in \mathcal{T}$.

For a future safety formula φ and TS $\mathcal{S} : \langle V, \Theta_{\mathcal{S}}, \mathcal{T} \rangle$,

T1. $\Theta_{\mathcal{S}} \rightarrow \theta_{\mathcal{A}}(\mathcal{A}(\varphi))$

T2. $\{\nu(n)\} \, \mathcal{T} \, \{\theta_{\mathcal{A}}(\delta(n))\}$ for $n \in \mathcal{N}(\mathcal{A}(\varphi))$

$\mathcal{S} \models \varphi$

Fig. 2. Basic temporal rule B-SAFE

Premise T1, the *Initiation Condition*, requires that the initial condition of $\mathcal{S}$ implies the initial condition of the automaton $\mathcal{A}(\varphi)$. Premise T2, the *Consecution Condition*, requires that for all nodes, $n \in \mathcal{N}(\mathcal{A}(\varphi))$, and for all transitions $\tau \in \mathcal{T}$, τ, if enabled, leads to the initial condition of the next-state automaton of n.

General Rule

As is the case with the rules B-INV and B-WAIT in [MP95], rule B-SAFE is hardly ever directly applicable, because the assertions labeling the nodes are not inductive: they must be strengthened. To represent the strengthening of an automaton, we add a new label μ to the definition of a node, $\langle \mu, \nu, \delta, f \rangle$, where μ is an assertion, and we change the definition of $\theta_{\mathcal{A}}$ for a node into

$$\theta_{\mathcal{A}}(\langle \mu, \nu, \delta, f \rangle) = \mu \ .$$

Using these definitions, Figure 3 shows the more general rule SAFE that allows strengthening of the intermediate assertions.

For a future safety formula φ, TS $\mathcal{S} : \langle V, \Theta_{\mathcal{S}}, \mathcal{T} \rangle$,
and strengthened automaton $\mathcal{A}(\varphi)$

T0. $\mu(n) \rightarrow \nu(n)$ for $n \in \mathcal{N}(\mathcal{A}(\varphi))$

T1. $\Theta_{\mathcal{S}} \rightarrow \theta_{\mathcal{A}}(\mathcal{A}(\varphi))$

T2. $\{\mu(n)\} \, \mathcal{T} \, \{\theta_{\mathcal{A}}(\delta(n))\}$ for $n \in \mathcal{N}(\mathcal{A}(\varphi))$

$\mathcal{S} \models \varphi$

Fig. 3. General temporal rule SAFE

Note that terminal nodes, that is, nodes with $\delta = \epsilon_{\mathcal{A}}$, never need to be strengthened. This is so, because consecution conditions from terminal nodes are all of the form $\mu(n) \wedge \rho_\tau \rightarrow \mathit{true}$, since $\theta_{\mathcal{A}}(\epsilon_{\mathcal{A}}) = \mathit{true}$, and thus trivially valid.

In [MS00] we show that rule B-SAFE is sound, that is, for a TS $\mathcal{S}$ and future safety formula φ, if the premises T1 and T2 of rule B-SAFE are $\mathcal{S}$-state valid then $\mathcal{S} \vDash \varphi$.

7 Implementation

Verification diagrams have been implemented in STeP, the Stanford Temporal Prover, a verification tool that supports algorithmic and deductive verification of reactive systems [BBC+95,BBC+00]. We are currently implementing support for interactive refinement and heuristics for automatic generation of verification diagrams.

The rule SAFE based on alternating automata has also been implemented in STeP, obviating the need for any specialized verification rules for safety properties. However, the strengthenings still have to be provided by the user.

Both verification diagrams and rule SAFE have been convenient in the proof of temporal properties, especially in proving properties of modular systems.

References

[BBC+95] N.S. Bjørner, A. Browne, E.S. Chang, M. Colón, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover, User's Manual. Technical Report STAN-CS-TR-95-1562, Computer Science Department, Stanford University, November 1995. available from `http://www-step.stanford.edu/`.

[BBC+00] N.S. Bjørner, A. Browne, M. Colón, B. Finkbeiner, Z. Manna, H.B. Sipma, and T.E. Uribe. Verifying temporal properties of reactive systems: A STeP tutorial. *Formal Methods in System Design*, 16(3):227–270, June 2000.

[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 *Lecture Notes in Computer Science*, pages 484–498. Springer-Verlag, 1995.

[BMS96] A. Browne, Z. Manna, and H.B. Sipma. Hierarchical verification using verification diagrams. In 2^{nd} *Asian Computing Science Conf.*, vol. 1179 of *Lecture Notes in Computer Science*, pages 276–286. Springer-Verlag, December 1996.

[Kur94] R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press, 1994.

[MBSU98] Z. Manna, A. Browne, H.B. Sipma, and T.E. Uribe. Visual abstractions for temporal verification. In A. Haeberer, editor, *Algebraic Methodology and Software Technology (AMAST'98)*, vol. 1548 of *Lecture Notes in Computer Science*, pages 28–41. Springer-Verlag, December 1998.

[MP87] Z. Manna and A. Pnueli. Specification and verification of concurrent programs by ∀-automata. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, number 398 in Lecture Notes in Computer Science, pages 124–164. Springer-Verlag, Berlin, 1987. Also in *Proc. 14th ACM Symp. Princ. of Prog. Lang.*, Munich, Germany, pp. 1–12, January 1987.

[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 *Lecture Notes in Computer Science*, pages 726–765. Springer-Verlag, 1994.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[MS00] Z. Manna and H.B. Sipma. Alternating the temporal picture for safety. In U. Montanari, J.D. Rolim, and E. Welzl, editors, *Proc. 27th Intl. Colloq. Aut. Lang. Prog.*, vol. 1853, pages 429–450, Geneva, Switzerland, July 2000. Springer-Verlag.

[Sip99] H.B. Sipma. *Diagram-based Verification of Discrete, Real-time and Hybrid Systems*. PhD thesis, Computer Science Department, Stanford University, February 1999. To appear as STAN-CS Technical Report.

[SUM99] H.B. Sipma, T.E. Uribe, and Z. Manna. Deductive model checking. *Formal Methods in System Design*, 15(1):49–74, July 1999. Preliminary version appeared in *Proc.* 8^{th} *Intl. Conference on Computer Aided Verification*, vol. 1102 of *LNCS*, Springer-Verlag, pp. 208–219, 1996.

[Tho88] W. Thomas. Automata on infinite objects. Technical Report 88-17, RWTH Aachen, 1988. In *Handbook of Theoretical Computer Science*, North-Holland.

[Var97] M.Y. Vardi. Alternating automata: Checking truth and validity for temporal logics. In *Proc. of the* 14^{th} *Intl. Conference on Automated Deduction*, vol. 1249 of *Lecture Notes in Computer Science*. Springer-Verlag, July 1997.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. First IEEE Symp. Logic in Comp. Sci.*, pages 332–344, June 1986.

Tracking Real-Time Systems Requirements

Aloysius K. Mok[†]
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
mok@cs.utexas.edu

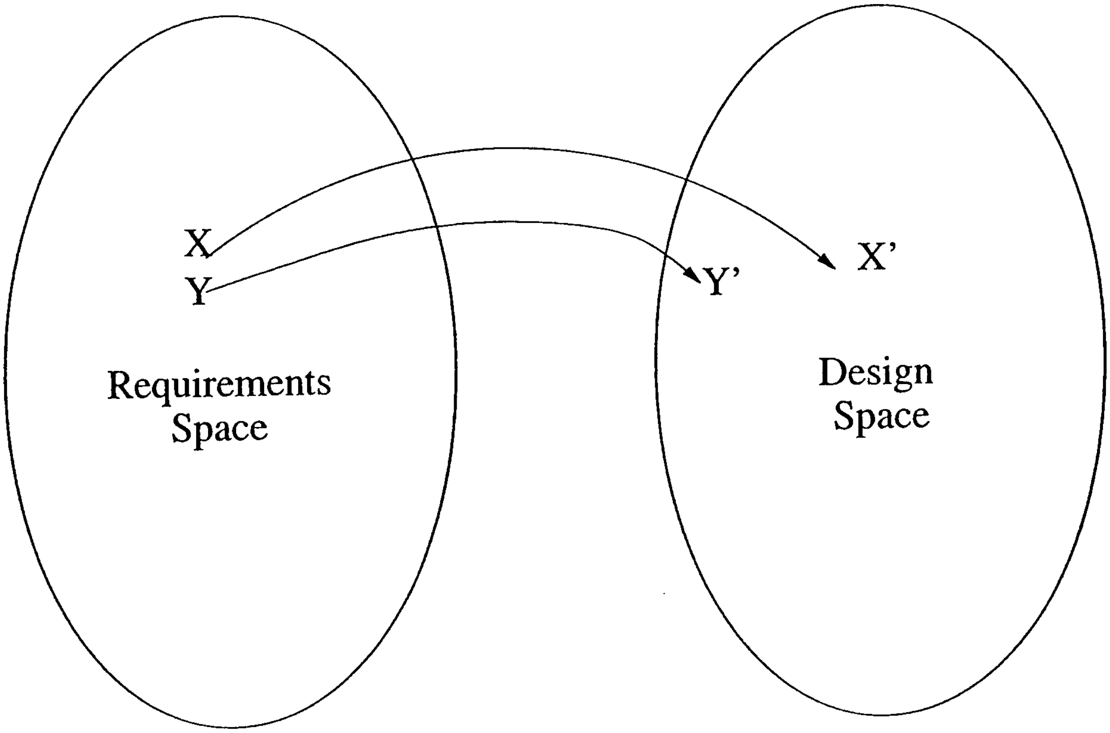
Abstract

One of the hard problems in maintaining real-time systems requirements is to keep track of the impact of resource usage on the applications. Often times, it is not sufficient to keep track of upper bounds since some requirements such as jitter are also sensitive to the lower bounds and the resource scheduling algorithm employed. In the case of non-preemptive scheduling, real-time requirements can be missed when resource utilization is decreased as in a CPU upgrade, even though there are no jitter constraints. In this paper, we define the notion of *robustness* for real-time performance requirments and discuss the tracking of sensitivity of real-time application requirements with respect to resource usage by formal method. We shall draw as an example the performance requirements of the avionics software of the Boeing 777 aircraft.

1. Introduction

A problem in engineering large complex software systems is the sensitivity of a design to changes in the requirements. If we view each step of the design process as a mapping from a requirements space to an (abstract) design space, the sensitivity problem may be viewed as a relation, let us call it the *tracking relation* between some appropriately defined *differnce metric* in the requirements space and the corresponding difference metric induced in the design space, where the difference metric measures the magnitude of some aspect of a change in the requirement/design space. There are some properties of the tracking relation that are obviously desirable. For example, the tracking relation should preserve *locality*: differences confined to a locality in the requirements space should induce differences confined to a locality in the design space, and *scalability*: a small difference in the requirements space should induce a small difference in the design space. Of course, how a difference metric is defined should reflect the aspect of requirements capture under consideration. For example, the difference metric meant to capture locality in the requirements space may reflect the number of functionalities/components that are affected by a change in the requirement, and the difference metric meant to capture scalability in the requirements space may reflect the increase in system load in a requirements change. The idea of tracking relation is illustrated in the following figure.

[†] The researh reported here is supported partially by a grant from the Office of Naval Research under contract number N00014-98-1-0704.



Want ||Y'-X'|| ~ ||Y-X||

Figure 1. Tracking Relation

In the following, we shall illustrate the tracking relation concept by considering a specific aspect of real-time systems design, specifically, the relation between a change in the real-time performance requirement and the schedulability of the design solution. Intuitively, if we make the real-time performance requirement of an application less stringent, we should expect the design solution to require at most the same amount of computing resources. A mapping from requirment to design is *robust* if a less demanding requirement will not cause a performance failure in the design. We shall formalize the concept of robustness in the context of real-time scheduling theory. The schedulability problem for preemptive schedulers was first discussed in [Liu&Layland 73]. We include below our own proofs for theorems 3 and 4.

2. Some Definitions

Let us first define some terminology. We shall assume that time is discrete and all timing parameters are integers. A slight modification is needed for the discussion below to apply to continuous time.

A *sporadic task* is charcterized by a pair: T_i=(C_i,P_i), where each request for service of T_i requires C_i units of CPU time to satisfy and two successive requests of T_i must be separated by at least P_i time units. Suppose M is a set of n *sporadic tasks* $\{(C_1, P_1), ..., (C_n, P_n)\}$ where C_i, P_i are respectively the computation time and the minimum separation between successive requests for sporadic task T_i. A preemptive fixed-priority (PFP) scheduler is used to schedule tasks in M. A PFP scheduler always selects for execution the task that has the highest priority. Unless otherwise stated, we shall adopt the convention that task T_i is assigned a higher priority than task T_j iff $i < j$. In the following, we talk about schedules for M that are produced by a PFP scheduler. We call these schedules PFP schedules.

Suppose r is a request for a task T_i that occurs at time t in a schedule s. Then the response time of r is defined to be $t' - t$ where t' is the time at which r is satisfied by the completion of the instance T_i in s corresponding to r. Given a priority assignment, a task is scheduleable iff all of its requests have response time no bigger than its minimum separation in every PFP schedule. A feasible schedule is one in which every task in the task set is scheduleable.

We can now define the robustness property as follows. The requirements space is the set of sporadic task sets. A design is a priority assignment to the tasks in a sporadic task set. Suppose we reduce the computation time or increase the minimum separation of a task in a given task set, we should expect the same design to work. In other words, a priority assignment that results in all the tasks in a task set being schedulable should preserve schedulability if the computation time is reduced or minimum separation increased for some task in the task set. We say that a priority assignment is *robust* if this is indeed the case. We shall see that the RMA priority assignment (defined below) is robust for the PFP scheduler.

3. The Robustness of PFP Priority Assignment

A task T_i is said to have an outstanding computation at time t in a schedule s iff a request r for T_i occurs at time t', $t' < t$, and r has not been satisfied at time t. Unless every request is satisfied before the arrival of the next request, it is possible for a task to have multiple outstanding computations at a time instant.

For a schedule s, a task T_i and time values t, t', define $N(i, s, t, t')$ to be the number of requests of T_i that appear in the interval $[t, t')$ in s. Notice that $N(i, s, t, t')$ is bounded from above by $\lceil (t' - t)/P_i \rceil$, and is exactly equal to $\lceil (t' - t)/P_i \rceil$ if a request for T_i occurs at t and T_i issues a request every P_i time units thereafter.

Lemma 1

Suppose s is a schedule such that there is no outstanding computation for any task at time t_0, for some $t_0 \geq 0$. Suppose a request r for task T_n occurs at t_0 in s, and r is satisfied at time t_1. Then the assertions (I1), (I2) below must hold.

(I1) $\forall\ t_0 < t < t_1,\ C_n + \sum_{1 \leq i < n} C_i \cdot N(i, s, t_0, t) > t - t_0$

(I2) $C_n + \sum_{1 \leq i < n} C_i \cdot N(i, s, t_0, t_1) = t_1 - t_0$

Proof:

Since there are no outstanding computations at time t_0 and the scheduler does not idle the processor whenever there is an outstanding computation, a simple induction on t shows that $C_n + \sum_{1 \le i < n} C_i \cdot N(i, s, t_0, t)$ is the amount of computation time that is needed to satisfy the request of T_n at t_0 plus the requests for all the higher priority tasks in the interval $[t_0, t)$ in s. QED

Lemma 2

Suppose s is a schedule such that there is no outstanding computation for any task at time t_0, for some $t_0 \ge 0$, and a request r for task T_n occurs at t_0. Let s' be a schedule in which there are no outstanding computations at time t_0, all tasks request simultaneously at time t_0, and all tasks request at their maximum rates thereafter. If the request r in s has a response time = u and the request for task T_n at time t_0 in s' has a response time = u', then $u' \ge u$.

Proof:

Let the completion times of the requests for T_n that occur at time t_0 in s and s' be respectively t_1 and t'_1. Assume the contrary: $u' < u$, i.e., $t'_1 < t_1$.

By applying assertion (I1) of lemma 1 to schedule s and assuming $t'_1 < t_1$, we have

$$C_n + \sum_{1 \le i < n} C_i \cdot N(i, s, t_0, t'_1) > t'_1 \text{-} t_0$$

Bounding $N(i, s, t_0, t'_1)$ by $\lceil (t'_1 - t_0)/P_i \rceil$, we have

$$C_n + \sum_{1 \le i < n} C_i \cdot \lceil (t'_1 - t_0)/P_i \rceil > t'_1 - t_0$$

Since all tasks request at their maximum rate from time t_0 in schedule s',

$$\forall\ 1 \le i < n,\ \ N(i, s', t_0, t'_1) = \lceil (t'_1 - t_0)/P_i \rceil$$

Substituting $N(i, s', t_0, t'_1)$ for $\lceil (t'_1 - t_0)/P_i \rceil$, we have

$$C_n + \sum_{1 \le i < n} C_i \cdot N(i, s', t_0, t'_1) > t'_1 - t_0$$

However, by applying assertion (I2) of lemma 1 to schedule s' where t'_1 is the completion time for the request at t_0, we have

$$C_n + \sum_{1 \le i < n} C_i \cdot N(i, s', t_0, t'_1) = t'_1 - t_0$$

Thus we have a contradiction. QED

Let M be a set of sporadic tasks. Let s be a PFP schedule of M in which all tasks request simultaneously at time 0, and all tasks request at their maximum rates thereafter. A task T in M is said to pass its *critical – instant test* if the response time of the request for T at time 0 in the schedule s is no bigger than its minimum separation parameter.

Suppose s' is a schedule in which there are no outstanding computations at time t_0, for some $t_0 \ge 0$, all tasks request simultaneously at time t_0, and all tasks request at their maximum rates thereafter. Since the schedule s and the suffix of s' after t_0 are identical, the response time of the request for a task T at time t_0 in s' does not exceed T's minimum separation parameter iff T passes its critical instant test.

Theorem 3 ([Liu&Layland 73])

If a task T passes its critical-instant test, then T is scheduleable by a PFP scheduler.

Proof:

Suppose T has the n^{th} highest priority (i.e., T is T_n and has a lower priority than tasks $T_1, \ldots, T_{n-1}$). We need to show that every request of T_n must have a response time no bigger than P_n in any PFP schedule. Let s be a PFP schedule. Without loss of generality, we shall disregard the scheduling of the tasks $\{T_i | i > n\}$, i.e., we consider only the n highest priority tasks. For any time value t_0 such that: (P1) there is no outstanding computation for T_n at time t_0 in s, and (P2) there is a request for T_n at t_0, we shall show that the request at t_0 must have a response time $\leq P_n$.

Let t_x be the biggest time value, $0 \leq t_x \leq t_0$ such that there is no outstanding computation for any task at time t_x. Since there is no outstanding computation at time 0, t_x must exist. Notice that if $t_x \neq t_0$, then the processor cannot idle in the interval $[t_x, t_0]$ and only tasks with priority higher than n are executed in $[t_x, t_0]$. Now consider a schedule s' such that there are no requests for the tasks $T_1, \ldots, T_{n-1}$ before t_x in s', and the requests for these tasks at or after t_x in s' occur at the same time as those in s. Also let the first request for T_n in s' occur at time t_x. By construction of s', the response time of this request is equal to $t_0 - t_x$ plus the response time of the request for T_n which occurs at t_0 in schedule s. Since there are no outstanding computations for $T_1, \ldots, T_n$ at t_x in s', and T_n passes its critical-instant test, the response time of the first request of T_n in s' must be at most P_n by lemma 2, and therefore the request for T_n at t_0 in s must have a response time not exceeding P_n.

The arrival time of the first request of T_n trivially satisfies (P1) and (P2), and hence the first request must have response time $\leq P_n$. Suppose the first i requests of T_n have response time $\leq P_n$. Then the $i+1^{th}$ request must satisfy (P1) and (P2), since the i^{th} and $i+1^{th}$ requests are separated by at least P_n time units. Hence the $i+1^{th}$ request must also have response time $\leq P_n$. QED

Given a task set M and a priority assignment, let s be the PFP schedule of M such that all tasks request simultaneously at time 0, and all tasks request at their maximum rates thereafter. We call s the critical schedule of M.

Corollary

If every task in a sporadic task set M meets its first deadline in the critical schedule of M, then M is scheduleable by a PFP scheduler.

Proof: Immediate.

The Rate Monotonic Assignment (RMA) of priorities:

Suppose $M = \{(C_1, P_1), \ldots, (C_n, P_n)\}$ is a set of n sporadic tasks, and task T_i has higher priority than task T_j if $i < j$. Then the priority assignment of tasks in M is consistent with RMA if $P_i \leq P_j$, $1 \leq i, j \leq n$.

Theorem 4 ([Liu&Layland 73])

Suppose M is a set of tasks whose priority assignment is consistent with RMA. Then M is scheduleable iff its critical schedule is feasible.

Unless otherwise stated, we shall refer to the critical schedule of a task set as one corresponding to a RMA-consistent assignment of priorities. For preemptive schedulers, reducing the computation time of a task in a scheduleable task set will not cause the resulting task set to be unscheduleable. To see this, suppose T has the n^{th} highest priority in the task set. Let the response time of the first request of T be x in s, the critical schedule, and let its response time be y in s', the critical schedule after the computation time of a higher priority task T_k has been reduced by some $\delta > 0$. If $y > x$, then applying assertion (I1) of lemma 1 to the critical schedule s' yields

$$C_n + \Big(\sum_{\substack{1 \le i < n \\ i \ne k}} C_i \cdot \lceil x/P_i \rceil \Big) + (C_k - \delta) \cdot \lceil x/P_k \rceil > x$$

Rewriting this inequality,

$$C_n + \sum_{1 \le i < n} C_i \cdot \lceil x/P_i \rceil - \delta \cdot \lceil x/P_k \rceil > x$$

However, applying assertion (I2) to the critical schedule s yields

$$C_n + \sum_{1 \le i < n} C_i \cdot \lceil x/P_i \rceil = x$$

which is a contradiction.

Similarly, it can be shown that increasing the period of any task in a scheduleable task set will not cause the resulting task set to be unscheduleable by a PFP scheduler. Thus, we have the following theorem.

Theorem 5

The RMA assignment of priorities for the PFP scheduler is robust.

In general, it can be seen that any feasible priority assignment for the PFP scheduler is robust. Unfortunately, this is not the case if the scheduler is non-preemptive.

4. Loss of Robustness in Non-preemptive Schedulers

In real-life systems, such as the Boeing 777 Integrated Airplane Information Management System (AIMS), not all tasks can be scheduled preemptively. The AIMS system, running on the ARINC 659 platform, requires high resource utilization and performance guarantees while providing strict partitioning of functions on a multiprocessor platform. The scheduling problem involves pre-scheduling of both computational and communication resources and involves both deadline and jitter requirements. A typical real-time requirement AIMS takes the form:

[Data] from <process> ([which runs at] <rate> <duration>) to <process>
of aggregate data transmission length <xfer duration> with minimum
latency <latency bound>.

Maximum latency <latency bound> means that the time from the start of execution of the sending process to the end of execution of the receiving process must not be less end than the specified bound. A +-500 usec jitter requirement (deviation from ideal period) across the board can be assumed, to both data and process start and end times.

Each of the above type of requirements involves three tasks, one application task for the sending process, one application task for the receiving process and one communication task for the transmission of data over the bus connecting the processors executing the application tasks. In total, there are 155 applications tasks and 951 communications between these tasks.

Whereas the application tasks may be preemptively scheduled on a processor, the communication tasks are inherently non-preemtpive as limited by the minimum size of a message. If we view the scheduling of messages on the bus as a single resource scheduling problem, a PFP scheduler is inappropriate for the communication resource. A non-preemptive fixed priority (NPFP) scheduler is one that always selects among all ready tasks the one that has the highest priority for execution until completion, i.e., once a task starts execution, no preemption is allowed, where a task is ready at time t if it has a request which arrives no later than time t and which has not been allocated execution time. We now show that the RMA priority assignment of NPFP is not robust.

Consider the following task set with 3 tasks: $\{T_1 = (3,5), T_2 = (2,10), T_3 = (4,20)\}$. With the RMA assignment, this task set is schedulable by a non-preemptive fixed priority scheduler as is shown by the timing diagram in the figure 2. However, the task set becomes unschedulable if we reduce the execution time of T_2 from 2 to 1. Hence, NPFP scheduler is not robust with respect to reduction of execution time requirement of a task.

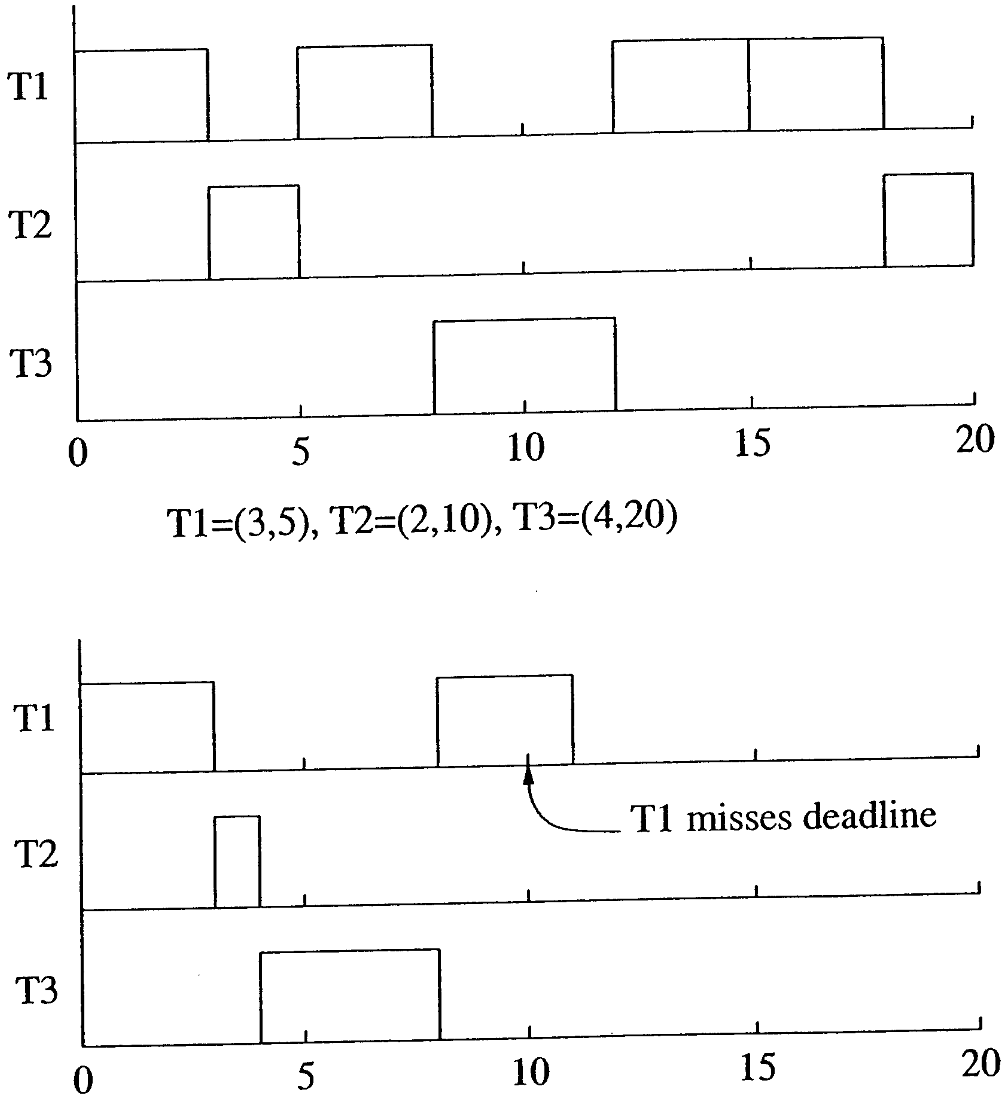
Next, consider the task set with 3 tasks: $\{T_A = (1,4), T_B = (3,8), T_C = (6,16)\}$. Again, with the RMA assignment, this task set is schedulable by a non-preemptive fixed priority scheduler as is shown by the timing diagram in the figure 3. However, the task set becomes unschedulable if we increase the period of T_A from 4 to 5. Hence, NPFP scheduler is not robust with respect to reduction of execution frequency of a task.

Lastly, consider the following task set with 3 tasks: $\{T_1 = (30,50), T_2 = (20,100), T_3 = (40,200)\}$. With the RMA assignment, this task set is schedulable by a non-preemptive fixed priority scheduler as is shown by the timing diagram in the figure 4. However, the task set becomes unschedulable if we reduce the execution times of all three tasks by 10%. Hence, NPFP scheduler is not robust with respect to improvement in CPU speed.

The above three counter-examples establishes

Theorem 6

The RMA assignment of priorities for the NPFP scheduler is not robust.

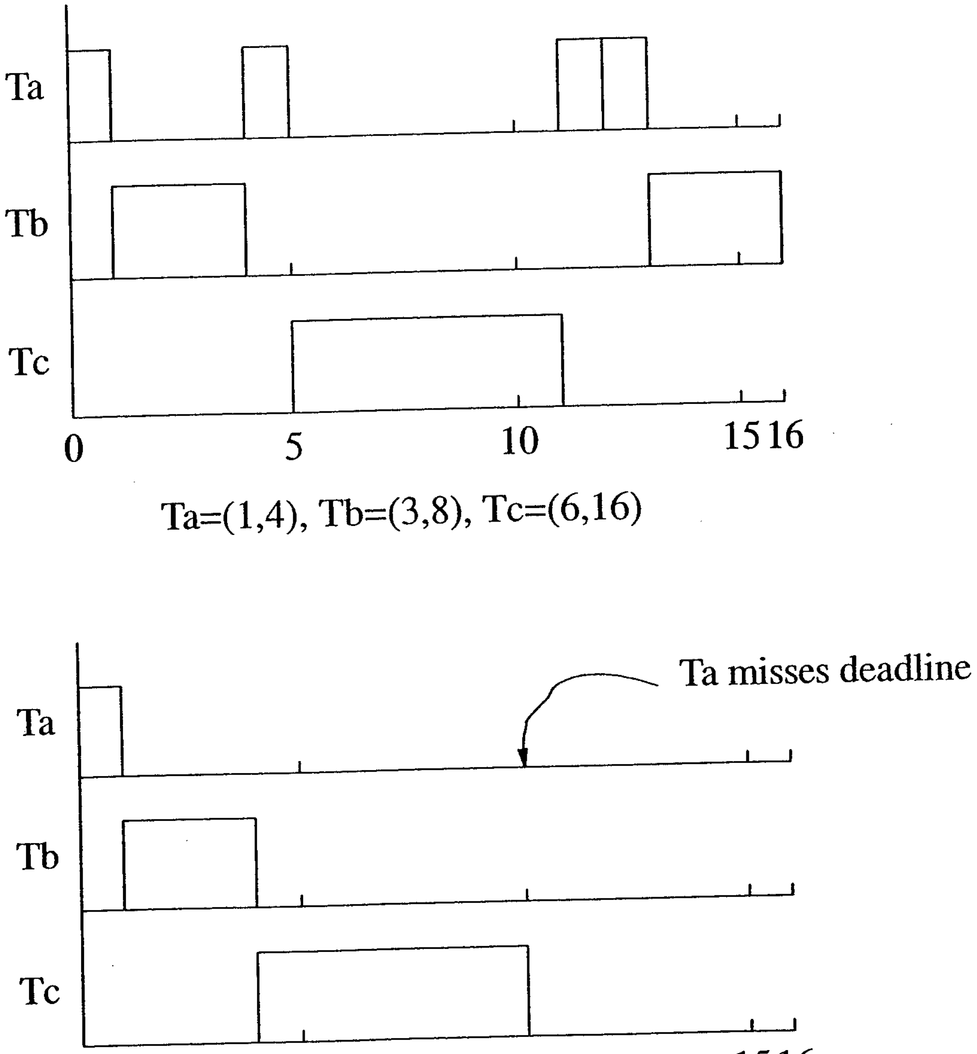


T1=(3,5), T2=(2,10), T3=(4,20)

T1=(3,5), T2=(1,10), T3=(4,20)

REDUCING C2 FROM 2 TO 1 CAUSES DEADLINE MISS

Figure 2. LOSS OF ROBUSTNESS WITH RESPECT TO DECREASED COMPUTATION TIME



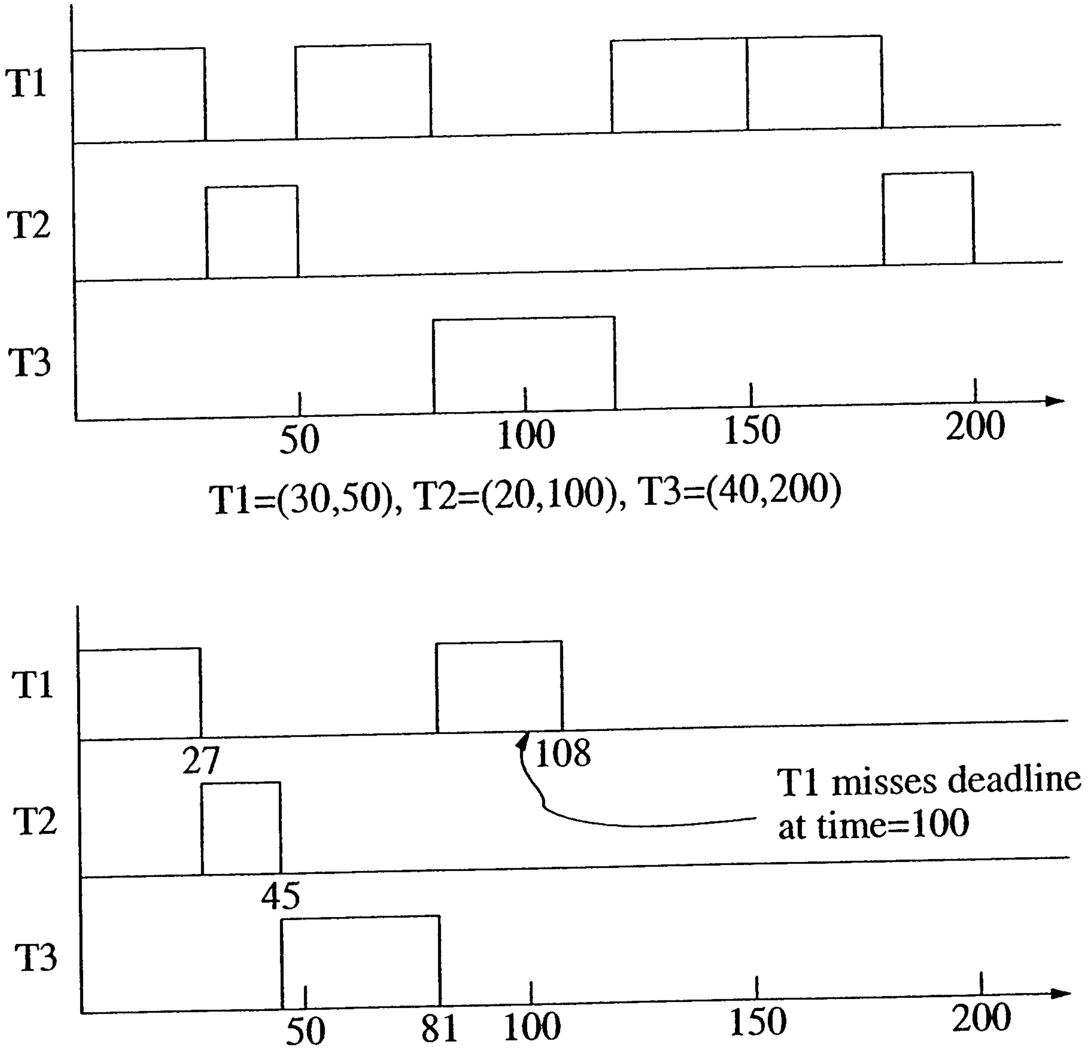
Ta=(1,4), Tb=(3,8), Tc=(6,16)

Ta misses deadline

Ta=(1,5), Tb=(3,8), Tc=(6,16)

INCREASING Pa FROM 4 TO 5 CAUSES DEADLINE MISS

Figure 3. LOSS OF ROBUSTNESS WITH RESPECT TO INCREASED PERIOD



T1=(30,50), T2=(20,100), T3=(40,200)

T1=(27,50), T2=(18,100), T3=(36,200)

FASTER CPU CAUSES T1 TO MISS DEADLINE
ALL EXECUTION TIMES DECREASED BY 10%

Figure 4. LOSS OF ROBUSTNESS WITH RESPECT TO IMPROVED CPU SPEED

The real-time scheduling solution adopted by the Boeing 777 AIMS relies on the use of *cyclic executives* which require the precomputation of static schedules which are then repeated at run time. In [MTR 96], we describe in detail a tool which automates the computation of the cyclic executives for requirements such as the Boeing 777 AIMS.

5. Conclusion

The robustness property discussed above is an example of the tracking relation in mapping requirements to design. Whereas the lack of robustness with respect to real-time performance requirements can be overcome by design automation tools such as the MSP.RTL tool described in [MTR 96], the more challenging problem is to achieve other properties such as locality and scalability simultaneously with performance robustness.

Bibliography

[Liu&Layland 73] C. L. Liu and James W. Layland, "Scheduling Algorithms for Multi-programming in a Hard-Real-Time Environment," *Journal of ACM, vol. 20, no. 1, January 1973.*

[MTR 96] A. K. Mok, Duu-Chung Tsou and R. C. M. de Rooij, "The MSP.RTL Real-Time Scheduler Synthesis Tool," *Proceedings of the 17th IEEE Real Time Systems Symposium, December 1966, pp. 118--128.*

Specification and Composition of Software Components: formal methods meet standards

CARLO MONTANGERO AND LAURA SEMINI

Dipartimento di Informatica
Università di Pisa
{monta,semini}@di.unipi.it
www.di.unipi.it/ ~{monta,semini}

Abstract. We describe how to use a combination of formal methods and standard middleware to approach COP with a coordination based attitude. Separating coordination from functionality, we foster the independent implementation of specific coordination templates on the middleware of interest. We discuss how a specific formal approach can be exploited to derive the interoperability skeleton in CORBA and C++ of the most common interaction template, i.e. client–server.

1 Introduction

Like in many other situations in software design, also in COP it is useful to divide and conquer, and both the component developer and the component integrator can gain from this attitude. The concerns to be separated are functionality and coordination. Indeed, coordination [1, 3] defines the ways components interact to reach the common goals when they are composed in a larger system, and can be largely independent from the specific functionalities of a given application. The separation is useful both at the high (specification) level and at the middle (interoperability) level. At the specification level, where the component integrator operates, separating functionality and coordination helps in understanding how to compose the components at hand. Indeed, there are well established coordination patterns, that can be studied per se, and lessen the needed cognitive load when attacking a new composition. On the other side, i.e. when developing components, coordination defines the structure that the middleware must implement to allow the components to interoperate correctly. Separating coordination from functionality, we foster the independent implementation of specific coordination templates on the middleware of interest. For instance, coordination may lead to the definition of skeletons in some middleware, like CORBA, to be completed by the code implementing the component functionalities. More in general, we can say that coordination defines the implementation of the interoperability level of a framework.

Formal methods can play a major role in COP. Precisely because the actors are programmatically independent, they need to have reliable ways to share precise knowledge of the artifacts they use or produce, independently of the particular technology (programming languages, middleware, ...) they are using. Formal methods offer exactly this kind of independence and precision, since they provide abstract models to share when operating with components. For instance, they provide ways to understand the potentialities and limitations of a coordination pattern, without the need to consider the specific middleware in use. Also, they can provide ways to make precise the specifications of the components and of their contextual dependencies, and to prove in advance global properties of a given composition, i.e. that the composition will meet the specifications it addresses.

An ideal world would see a unique universal middleware for interoperability in COP. In the real world the situation is muddier, but a number of standards (CORBA, COM, DCOM) are emerging, and it is natural to consider them as targets of component development.

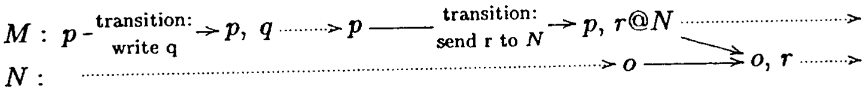
This paper describes how we are using a combination of formal methods and standard middleware to approach COP with the coordination based attitude described above. We discuss how a specific formal approach, Oikos–*adtl* [4], can be exploited to derive the interoperability skeleton in CORBA and C++ of the most common interaction template, i.e. client–server.

We first briefly present the abstract computational model and the temporal logic system to reason on abstract computations. We then outline how components and coordination template can be specified and verified. Finally, we sketch how the skeleton middleware can be systematically derived and justified from the logical specification.

2 Background: Oikos–*adtl* and composition templates

Oikos–*adtl* is a specification language for distributed systems based on asynchronous communications. The language is designed to support the composition of specifications. It allows expressing the global properties of a system in terms of local properties and of *composition templates*. The former are properties exposed by a single component, the latter describe the approach taken to control the interactions of the components. Oikos–*adtl* is based on an asynchronous, distributed, temporal logic, which extends Unity [2] to deal with components and events.

The Computational Model. A system $\mathcal{T}_{\mathcal{M}}$ is characterized by a list of components $\mathcal{M} = M, N, O \ldots$, and a set $\mathcal{T}$ of state transitions. A computation is a graph of states like the one in the figure below.



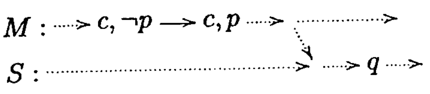
Syntax and Semantics. A specification shapes $\mathcal{M}$:≪ $\mathcal{F}$ ≫, where $\mathcal{M}$ is a set of component names and $\mathcal{F}$ is a set of formulae. Formulae describe computations relating *state formulae*, which express properties of single computation states. Formulae are built using the temporal operators CAUSES_C, CAUSES, NEEDS, BECAUSE_C, BECAUSE, WHEN, and those of Unity. They shape like:

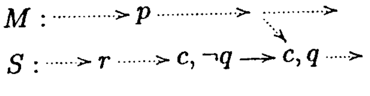
$$M\!:\!p \text{ WHEN } c \text{ CAUSES } S\!:\!q \quad (1)$$

$$S\!:\!q \text{ WHEN } c \text{ BECAUSE } (S\!:\!r \wedge M\!:\!p) \quad (2)$$

$$M\!:\!p@O \text{ CAUSES\_C } O\!:\!p \quad (3)$$

With *event p* we mean that property *p becomes* true. (1) reads: event *p* in *M* when *c* holds causes condition *q* in *S* and states that any state of *M* in which *c* holds and *p* becomes true, is eventually followed by a state of *S* satisfying *q*, as in the computations to the left, below. Formula (2) states that event *q* can occur in *S* when *c* holds only if previously *r* and *p* have hold in *S* and *M*, respectively, like in the computation to the right, below.





The state formula $M : p@O$, means that component M wants to send p to O: (3) is an axiom for Oikos–*adtl*. It says that messages are immediately sent and are guaranteed to arrive. Suffix c stands for *closely*: for instance, in (3), CAUSES_C requires messages to be sent immediately. Operator BECAUSE_C requires that the condition enabling the event happened in the same or in the previous state. NEEDS requires a condition to hold when an event occurs.

Composition Templates. To specify and prove the global coordination properties of a distributed system is often a complex task. A coordination template defines a composition schema for a set of components: the global properties of their parallel composition can be obtained as theorems. Templates shape:

$$\begin{array}{ll} & M^1, \ldots, M^n :\ll \mathcal{F} \gg \\ \sqsubseteq & \dfrac{\vdash_{M^1,\ldots,M^i} \mathcal{F}' \quad \vdash_{M^{i+1},\ldots,M^n} \mathcal{F}''}{\vdash_{M^1,\ldots,M^n} \mathcal{F}} \\ & M^1, \ldots, M^i :\ll \mathcal{F}' \gg \parallel M^{i+1}, \ldots, M^n :\ll \mathcal{F}'' \gg \end{array}$$

The schema above shows the typical structure of our composition theorems, and reads: the parallel composition of any pair of systems $\mathcal{T}'_{M^1,\ldots,M^i}$, and $\mathcal{T}''_{M^{i+1},\ldots,M^n}$ satisfying $\mathcal{F}'$ and $\mathcal{F}''$ resp., satisfies $\mathcal{F}$. A system satisfies the set of formulae $\mathcal{F}$ iff all its computations are models of all the formulae in $\mathcal{F}$.

The proofs are sequences of applications of composition templates, and show a general pattern: first (reading them bottom–up) lift local properties to the global level, and then apply transitivity or other composition rules.

A large set of composition rules are available to reason on Oikos–*adtl* specifications. To give a flavor, we list the most useful ones. (5) is a lifting rule for $\textsc{because\_c}$: the new component can be an unforeseen cause for A. (4) is a transitivity result for $\textsc{causes}$.

$$\frac{\vdash_{\mathcal{M}} M{:}A \;\textsc{causes}\; O{:}C \quad \vdash_{\mathcal{M}} O{:}C \;\textsc{causes}\; N{:}B \quad \vdash_{\mathcal{M}} O{:}C \;\textsc{because}\; M{:}A}{\vdash_{\mathcal{M}} M : A \;\;\textsc{causes}\;\; N : B} \tag{4}$$

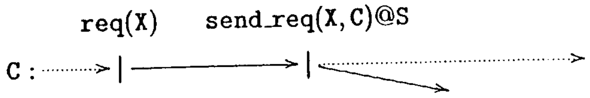
$$\frac{\vdash_{\mathcal{M}} \;\; M : A \;\textsc{because\_c}\; N : B}{\vdash_{O,\mathcal{M}} \;\; M : A \;\textsc{because\_c}\; (N : B \;\vee\; O : A@M)} \tag{5}$$

3 Using Composition Templates

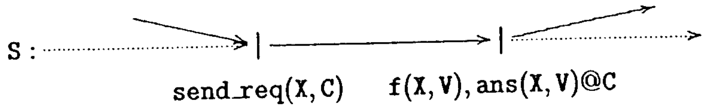
A simple example of composition template is the client–server template, which is still one of the most common approaches to component composition: we used it to try out our approach. The template shows the local descriptions of client C and server S, and derives the properties of their interaction:

$$\begin{array}{lll} C,S : & \ll C : req(X) \;\textsc{causes}\; C : ans(X,V) & (6) \\ & C : ans(X,V) \;\textsc{because}\; S : f(X,V) & (7) \\ & C : ans(X,V) \;\textsc{because}\; C : req(X) \quad \gg & (8) \\ \sqsubseteq & & \\ C : & \ll C : \;\textsc{inv}\; server(S), & (9) \\ & C : req(X) \;\textsc{when}\; server(S) \;\textsc{causes\_c}\; C : send\_req(X,C)@S, & (10) \\ & C : send\_req(X,C)@S \;\textsc{needs}\; req(X), & (11) \\ & C : ans(X,V) \;\textsc{needs}\; false, & (12) \\ & C : f(X,V)@S \;\textsc{needs}\; false \quad \gg & (13) \\ \parallel & & \\ S : & \ll S : send\_req(X,C) \;\textsc{causes\_c}\; S : ans(X,V)@C, & (14) \\ & S : ans(X,V)@C \;\textsc{because}\; S : f(X,V), & (15) \\ & S : f(X,V) \;\textsc{because}\; S : send\_req(X,C), & (16) \\ & S : send\_req(X,C) \;\textsc{needs}\; false \quad \gg & (17) \end{array}$$

Following [5], a server component is characterized by the fact that a request to the server carries explicitly the name of the client, in order to deliver correctly the response. The client knows the server (9), sends it its requests (10, 11), and does not produce (12) or compute (13) on its own answers to the requests. Computations of the client look like:

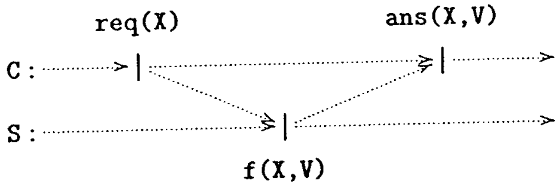


The server answers the requests of the client (14). Answers are produced after computing some value V (15), and this is done only upon a request (16). Finally, servers do not produce requests (17). Computations of the server look like:



The local specifications fix the interface of the server, and the most abstract functional constraint, i.e. that a value has to computed by the server before answering.

The components can be developed independently, since the pattern, having fixed the interactions, projects on each component its responsibility. Besides, the pattern proves in advance that the interaction between client and server satisfies some global properties: requests will receive an answer (6); answers respect the values computed by the server (7) and are received only upon request (8). The following picture shows a computation satisfying these properties.



Our case study is taken from the CORBA documentation, and considers a simple Stock Exchange. The system is composed of a *Quoter* and a set of clients Ci. Clients interact with the Quoter to be informed on the values of some stocks at the Stock Exchange. In particular, the generic client Ci interacts with the Quoter if interested in the value of stock X ($Stock\_value(X)$). The Quoter computes the current value V for X (($current(X,V)$), and sends the data to the client ($quote(X,V)$).

$$\begin{array}{rl} Quoter, Ci : \ll & Ci : stock\_value(X) \text{ \textsc{causes} } Ci : quote(X,V), \\ & Ci : quote(X,V) \text{ \textsc{because} } Quoter : current(X,V), \\ & Ci : quote(X,V) \text{ \textsc{because} } Ci : stock\_value(X) \quad \gg \end{array}$$

The example shows a client–server interaction among the clients and the Quoter. We thus instantiate the client–server template and obtain the following local specifications, where $get\_quote(X,Ci)$ encodes the request of Ci:

$$\begin{array}{lll} Quoter : \ll \text{(for all i)} & & \\ & Quoter : get\_quote(X,Ci) \text{ \textsc{causes\_c} } Quoter : quote(X,V)@Ci, & (18) \\ & Quoter : quote(X,V)@Ci \text{ \textsc{needs} } current(X,V), & (19) \\ & Quoter : current(X,V) \text{ \textsc{needs} } get\_quote(X,Ci), & (20) \\ & Quoter : get\_quote(X,Ci) \text{ \textsc{needs} } false \quad \gg & (21) \end{array}$$

$$
\begin{array}{lll}
Ci & :\ll Ci : \textsc{inv}\ server(Quoter), & (22)\\
 & \quad Ci : Stock\_value(X)\ \textsc{when}\ server(Quoter)\ \textsc{causes\_c} & \\
 & \qquad\quad Ci : get\_quote(X, Ci)@Quoter, & (23)\\
 & \quad Ci : get\_quote(X, Ci)@Quoter\ \textsc{needs}\ Stock\_value(X), & (24)\\
 & \quad Ci : quote(X, V)\ \textsc{needs}\ false, & (25)\\
 & \quad Ci : current(X, V)@Q\ \textsc{needs}\ false \quad \gg & (26)
\end{array}
$$

4 Implementing a coordination template

The abstract coordination template described above can be transformed once for all in a concrete interaction skeleton in a standard middleware. We exemplify this with CORBA and C++. Given that our logic setting is asynchronous, we use the recent specification of CORBA asynchronous method invocation (AMI, [6]).

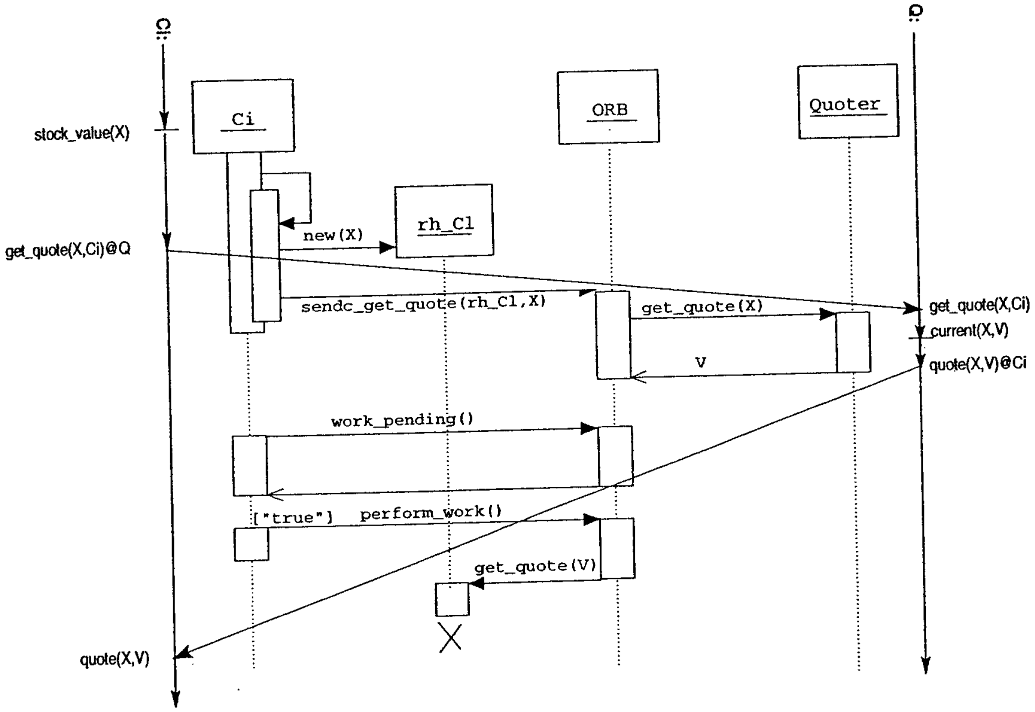


Fig. 1. Correspondence template–CORBA

AMI has been conceived so that no changes are needed, with respect to the synchronous case, on the side of the callee (the server side). Is is up to the caller (the client) and to the ORB to cope with the differences with synchronous invocations. This is apparent in the UML sequence diagram in Fig. 1: the interactions on the left side (client-ORB) are more complex than those on the right side (ORB-server). The figure uses the Stock Exchange example to illustrate the correspondence between the computation template and CORBA. The server interface

```
interface Quoter {
   double get_quote(in string X);
}
```

can be derived systematically from the logical specification. The CORBA compiler generates also a skeleton class implementing the server:

```
class Quoter_i : public virtual POA::Quoter {
  ...
public:
  ...
   CORBA::Double get_quote(const char* X) {
      CORBA::Double V;
      \\ insert here the code to compute V,
      \\ such that current(X, V) holds
      return V;
   };
};
```

As the superposition of the client–server template on the sequence diagram in Fig. 1 should make clear, what is left on the server side is the implementation of the server functionality, as specified by predicate *current*.

From interface Quoter, the CORBA compiler generates also a stub C++ class, which exports a method void sendc_get_quote(AMI_Quoter_Handler, char*) that the client uses to invoke the Quoter method get_quote(char*) remotely via the ORB. The second argument of the asynchronous call is the C++ transposition of the original argument to get_quote, and is delivered to the server by the ORB, via the server method. The first argument is a callback handler created by the client. This object exports a method get_quote(double) that the ORB exploits to return the answer asynchronously.

The skeleton of the callback handler can also be generated from the specification:

```
class Handler : public POA::AMI_Quoter_Handler {
private:
    CORBA::X_type X ;
    // initialized to the original argument to get_quote
..     .
public:
    void req(CORBA::V_type V) {
    // insert here the continuation code of the client:
    // quote(X,V) has been established
    };
    ...
};
```

The client is not blocked, and uses the ORB methods work_pending() and perform_work() to control when to receive the answer via the handler.

The destructor of the client object can be defined to implement the busy–waiting cycle that is necessary, according to the AMI specification, to force the ORB to deliver the answer from the server, when available. In the simplest cases, this allows the client code to terminate immediately after the asynchronous call, and let the handler complete the computation, once the answer is available. This is suggested by the comment in the code above. In other cases, e.g. when the client has to exploit several answers, it may be necessary to split the code between the client and the handler: this may require some refinement steps, to decide how to proceed. How much of these refinements can be standardized is still a matter of investigation.

5 Conclusions

We think that our approach shows that formal methods can play an essential role in characterizing component coordination at the abstract level, identifying the interactions between components and their context, to a point where standard skeletal implementations can be rigorously derived, for a large set of standard middleware. This can liberate component oriented programming from the burden of repetitive tasks, leaving space to more ingenious activities, related to the specifics of the problem at hand.

Acknowledgements

This work was partly supported by the ESPRIT W.G. 24512 COORDINA and the Italian MURST project SALADIN. R. Dolfi experimented with CORBA. D.C. Schmidt and T. Flagella offered helpful support (remotely and locally, respectively).

References

1. N. Carriero and D. Gelernter. Coordination Languages and their Significance. *Communications of the ACM*, 5(2):97–107, 1989.
2. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading Mass., 1988.
3. P. Ciancarini and A. Wolf, editors. *Proc. 3nd Int. Conf. on Coordination Models and Languages, COORDINATION 99*, volume 1594 of *Lecture Notes in Computer Science*, Amsterdam, April 1999. Springer-Verlag.
4. C. Montangero and L. Semini. Composing Specifications for Coordination. In *[3]*, pages 118–133.
5. C. Montangero and L. Semini. Refining by Architectural Styles or Architecting by Refinements. In L. Vidal, A. Finkelstein, G. Spanoudakis, and A.L. Wolf, editors, *2nd International Software Architecture Workshop, Proceedings of the SIGSOFT '96 Workshops, Part 1*, pages 76–79, San Francisco, CA, Oct 1996. ACM Press.
6. OMG. CORBA Messaging Joint Revised Submission. Technical Report orbos/98-05-05, Object Managemnet Group, Framingham, MA, 1998.

Exploiting formal methods in the real world: a case study of an academic spin-off company

G. M. Reed

Oxford University and Formal Systems

There have many academic spin-off companies set up over the past decade to exploit fundamental research in formal methods. Very few of these companies have survived, and even fewer have made significant profits. In this paper, as co-founder and director of one such company, I will give a brief history of the company, and discuss the challenges and opportunities involved.

1 Formal Systems

Formal Systems (Europe) Ltd was founded in Oxford in 1989 to exploit the research on CSP and Z done at Oxford University in the 80's. This research had high international recognition, and very strong links to UK industry. There is a very prestigious UK award (the Queen's Award for Technological Achievement) which is essentially an institutional "knighthood". In 1990, Oxford University and Inmos Ltd won this award for the development of formal methods in the specification and design of microcomputers; this work involved CSP and the occam programming language. The award cited that the use of formal methods had reduced the development time of the IMS T800 Transputer by 12 months. In 1992, Oxford University and IBM won the award for the use of formal methods - notations, theories, and processes, and specifically the use of the Z notation - in the production of the CICS transaction processing software. The award cited that formal methods had reduced the development cost by an estimated half million dollars. For each of these two awards, the Oxford team leaders (Bill Roscoe and Jim Woodcock) became directors of Formal Systems, and several of the team members became full-time employees (in particular, Michael Goldsmith became managing director).

During the early 90's, Formal Systems developed the FDR model-checker for concurrent finite state machines. FDR was developed as a result of collaborative work between Formal Systems and Inmos Ltd as a tool for verifying VLSI designs. It was used extensively in the design of the T9000 transputer and the C104 routing chip. Subsequently, it was used in the design and verification of a fault-tolerant processing system for a high-reliability embedded control system in conjunction with Draper Laboratories in the US. FDR found previously undetected faults in the prototype developed by Draper, and was used in a redesign of the software. This redesign was proved correct, and the final code was reduced 75 percent from that of the original.

Given the above start, it was assumed that Formal Systems would quickly grow into a large profitable company. Although successful, this growth has not yet happened. I will indicate below some of the lessons we have learned, and indicate some new commercial strategies.

2 Problems

2.1 Marketing problems with customers

- reluctance to use new notation
- confusion over multitude of formal methods
- level of ability and experience needed to understand and design models simply too high
- correctness of software not crucial; post-production errors can be patched over internet
- formal methods seen as causing unacceptable delay in the development process
- contact with customer is via their research department which comes to view you as a competitor for resources

2.2 Technical problems

- methods often do not scale
- state-explosion in model-checking
- difficult to make tool support usable for practitioners

2.3 Internal problems

- good researchers are often not good at marketing
- resistance of academics to a change of culture
- temptation to become subdepartment of university

3 Some solutions

3.1 Simplify notations and combine methods

Most of the current work at Formal Systems involves the use of FDR, which explores the behaviour of a pair of processes, using the operational semantics of CSP to determine whether one process (the implementation) refines another (the specification) or, if not, to find a counterexample by producing a behaviour of the implementation not found in the specification. Using the same language for describing both systems and properties greatly simplifies stepwise refinement and compositional development (see [R]). It also presents the user with only one notation to master.

It is clear that some system properties are best described in terms of state changes and others in terms of communications. It is also clear that automated verification, if possible, is sometimes more efficient via theorem-provers and sometimes more efficient via model-checkers ([RSR],[RSG]). Using action systems, it is possible to give a state-based approach within the semantics of CSP ([B],[M]), and thus use FDR to check state-based properties. Recent work in [RS] gives a technique for coupling specification and verification of components within a system using state-based specification and theorem-proving on some components and event-based specification and model-checking on others. This work is based on the theory of refinement in the semantic models of CSP.

Recent work [OR] has also shown how the semantics of (real-time) Timed CSP [RR] can be translated into a discrete timed model for CSP in which properties can be model-checked on FDR. Hence, it is now possible to specify and verify temporal behaviour within the same notation.

3.2 Attack state-explosion and scaling problem

New compression techniques developed by Formal Systems over the past few years together with increased processing power has significantly raised the number of states which can be explored. It is now practical to explore $O(10^8)$ states at 10^7 per hour on standard hardware. However, this is still not sufficient for many real-world problems.

Real advances to the problem are coming from the work in ([Rl],[SRe],[SRo]) on data independence and induction. The work in [RL] gives methods to calculate automatically thresholds for model parameters such as size of data types, number of nodes, etc. If a model is instantiated with parameters of at least the size of the thresholds and a property is proved correct, then the property is established correct for all values of the parameter. New induction techniques in [SRe] and combined with data-independence results in [SRo] are also proving powerful techniques in the reduction of the state space. These techniques allow FDR to formally establish properties of arbitrary branching networks.

3.3 Identify the most advantageous problem domain

As mentioned above, the original application domains for Formal Systems and FDR were in the design and verification of VLSI and embedded control systems. It was only when noticing that a large percentage of the audience at an industrial course offered by Formal Systems in Boston were from "security" agencies, that it became evident it might be beneficial to produce applications in the domain of computer security. The resulting applications to security protocols and information flow developed by Roscoe and Woodcock have since achieved an almost benchmark status for applying model-checkers.

Currently, eighty percent of Formal System's contracts are in the domain of computer security and are funded by UK and US agencies. We are now moving into the commercial security market.

3.4 Hide the formal methods from the customers

As noted, there is considerable resistance to the use of formal methods by potential customers. One strategy is to hide the pain and only show the benefits.

In collaboration with GrammaTech Inc., and with funding from the US Office of Naval Research, Formal Systems is engaged in a two-year project aimed at property-checking in the UML context. The goal is to exploit state-machines which are created by the Rational/ObjecTime Rose RealTime tool [RSC]. Such state-machines are currently relegated to a documentary role. By applying FDR behind the scenes, it is possible to provide valuable information to the user without reguiring their explicit knowledge of FDR [WRG].

Formal Systems is also currently negotiating with Motorola and Telelogic in the joint development of a verification tool with SDL as the specification language with a translation compatible to FDR. Such a tool would be of considerable use in telecommunications.

Gavin Lowe has developed the tool Casper [L] which allows security protocols to be specified in the usual ASCII-fied script found in the security literature, and then be translated into a script for verification by FDR. Hence, security protocols can be verified by people with no previous experience of model-checking.

A new commercial strategy in the use of formal methods by Formal Systems is to establish a "bank" for verified software. Software in the bank has been specified in conjunction with its owner and (to the state-of-the-possible) verified by Formal Systems. It is held for reusability, updating, and legal proceedings. It is not necessary for the owner to know formal methods. Once software is in the bank, it becomes available as a "component" to be used with only the necessity of verifying the connections between other components. The use of formal methods is completely hidden from the customer, but provides a clear benefit.

3.5 Bring in the suits

Finally, it is probably necessary to take on venture capital, buy in management and marketing, and become greedy capitalists. Once over the culture shock, we can talk about the good old innocent days while drinking marguerites in the Caribbean.

4 Acknowledgements

Much of the technical background for this paper was taken from [G] and [Z]. The reader should consult these papers for more detail.

5 References

[B] M.J. Butler, *A CSP approach to action systems*, DPhil thesis, Oxford University, 1992.

[CRe] S. Creese and J.N. Reed, *Verifying end-to-end protocols using CSP/FDR*, Proceedings of IPPS/SPDP Workshop on Parallel and Distributed Processing, LNCS 1586, Springer, 1999.

[CRo] S. Creese and A.W. Roscoe, *Formal verification of arbitrary network topologies*, Proceedings of PDPTA'99, CSERA Press, 1033-1039.

[G] M. Goldsmith, *Challenges to process-algebraic property-checking*, Proceedings of PDPTA'99, CSREA Press, 273-278.

[L] G. Lowe, *Casper: a complier for the analysis of security protocols*, Proceedings of the 10th IEEE Computer Security Foundations Workshop, 1997.

[LR] R. Lazic and A.W. Roscoe, *Data indepencence with generalised predicate symbols*, Proceedings of PDPTA'99, CSREA Press, 319-325.

[M] C.C. Morgan, *Of wp and CSP*, Beauty is our business: a birthday salute to Edsger W. Dijkstra, editors: D. Gries, W.H.J. Feijen, A.G.M. van Gasteren, and J. Misra, Springer-Verlag, 1990.

[OR] J. Ouakine and G.M. Reed, *Model-checking temporal behaviour in CSP*, Proceedings of PDPTA'99, CSREA Press, 295-304.

[R] A.W. Roscoe, *The Theory and Practice of Concurrency*, Prentice Hall, 1998.

[RR] G.M. Reed and A.W. Roscoe, *The timed failures-stability model for CSP*, Theoretical Computer Science 211 (1999), 85-127.

[RS] J.N. Reed and J.E. Sinclair, *Bicompositional refinements of loosely coupled specifications*, submitted for publication.

[RSG] J.N. Reed, J.E. Sinclair, and F. Guigand, *Deductive reasoning versus model checking: two formal approaches for system development*, Proceedings of Integrated Formal Methods 99, 1999.

[RSR] J.N. Reed, J.E. Sinclair, and G.M. Reed, *Routing: a challenge to formal methods*, Proceedings of PDPTA'99, CSERA Press, 305-311.

[WRG] P. Whittaker, G.M. Reed, and M. Goldsmith, *Formal methods adding value behind the scenes*, Proceedings of PDPTA'2000, CSERA Press.

[Z] I. Zakiuddin, *Current limits for exploiting automated verification*, Proceedings of PDPTA'99, CSERA Press, 319-326.

Experimental Analysis for Large Agent Systems

Dave Robertson
Division of Informatics, University of Edinburgh
D.Robertson@ed.ac.uk

This extended abstract gives a basic introduction to the aims of a new project, funded by the European Commission, on the experimental analysis of large multi-agent systems. The project involves the University of Edinburgh (Anderson, Fourman, Robertson, Sannella, Vasconcelos, Walton), the Instute for Artificial Intelligence in Barcelona (Agusti, Sabater, Sierra) and the University of Liverpool (Parsons, Wooldridge).

Engineers working on large, distributed, multi-agent systems face a problem which differs from conventional software engineering. They build software systems which must coexist with other systems about which little may be known, yet we wish the overall behaviour of the population of systems to be predictable in certain ways depending on the domain of application. There is a loose analogy to the biosciences, where the scientific response to the problem of understanding population behaviour has been to build mathematical models at various different levels of granularity of detail and use these to help form hypotheses about the driving forces in very complex ecological systems. We are beginning to follow a similar path in analysing multi-agent systems.

Two technical prerequisites for solving this problem are a framework within which to design and run experiments on models of large agent systems and clear software engineering methods which allow the results of experimental analyses to be related to agent design choices. These currently do not exist. There are numerous agent deployment systems but no convincing systems for modelling agent populations and their evolution. There are numerous software engineering methods but none of these translate easily to agent design. Providing a combination of analysis and design in this area must therefore be a ground-up exercise, drawing upon fragments of research from related areas.

Section 1 gives an overview of what we hope to achieve by the end of the project. Section 2 gives a flavour of the sort of analysis by working through a basic example constructed in the first weeks of the project.

1 What we Hope to Achieve

An overview of our proposed framework is shown in Figure 1. In the centre of the diagram is a laboratory system in which experiments are run on multi-agent models. Two major processes surround the laboratory. To the left is the design of agents which is constrained by design rules (determining key features agents can have) and hazard analyses (warnings of threats created by design choices). These controls on design are extended and revised in the light of analyses from laboratory experiments, allowing us to feed empirical results relating to system behaviour back into the definition of design controls - a "virtuous cycle" of design → experiment → redesign. The second major process (to the right) develops an overall theory, for our experimental systems, of the response of ecosystem properties to design decisions. This influences the conduct of subsequent experimental analyses, which are used to reinforce and extend the experimental theory. It is also compared to case studies from real-world systems which either confirm the experimentally observed behaviour or generate exceptions to it. These exceptions prompt theory revision, giving a second "virtuous cycle" of experiment → theorise → validate → re-theori se → re-experiment.

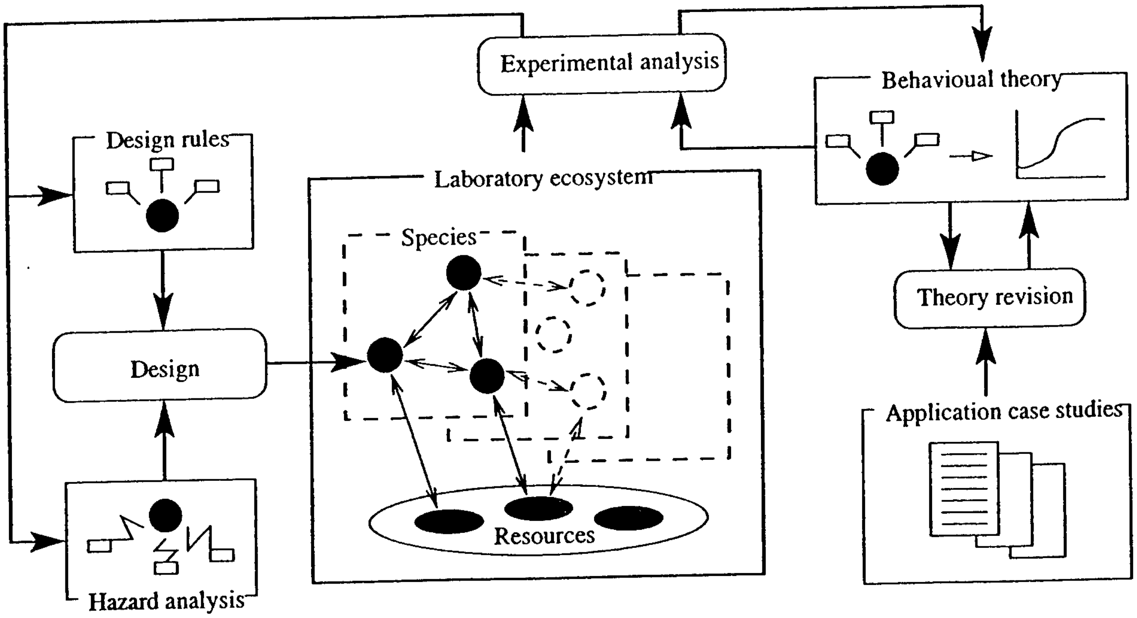


Figure 1: Conceptual overview of the project

Our aim is to innovate in the design lifecycle and validate with respect to ecosystem lifecycles. We are interested in relating the methods we use to design ecosystem inhabitants to aggregate behaviours observed at the ecosystem level.

2 Introductory Example

The purpose of this section is to give a straightforward example of analysing a rudimentary agent system. First we describe the system to be modelled; then describe the model itself; then perform some experiments with the model; and finally show how a potential threat occurring in the model may be countered.

2.1 The System to be Modelled

In the financial industries we now find systems where some centrally held resource cannot in practice be distributed directly from the centre but must be distributed via mediators. To allow consumers of the resource to be dealt with promptly, each mediator has its own system of bookkeeping and authority from the centre to decide which consumers to supply. Messages passed back to the centre allow the overall pattern of resource use to be monitored.

An example of this type of system is in bank account transactions. The resource in this case is the money in the bank. Its consumers are customers with bank accounts. The mediators are the points of contact for account withdrawals: at a branch, through a telephone call centre or via an internet site. Each of these points of contact may keep its own database of customer account details and these frequently will be reconciled with one or more central accounts databases.

2.2 The Model

The agents in the model and their potential interactions are illustrated in Figure 2. We can have any number of central resource agents (labelled R on the illustration). Each of these has some number of mediators (labelled M) and we shall assume for simplicity that this number is the same for each central resource. There can be any number of consumers (labelled C) and each may interact with one or more groups of mediators (in the illustration, C1 and C2 interact with the mediators for R1 while C3 interacts with the mediators for R1 and R2).

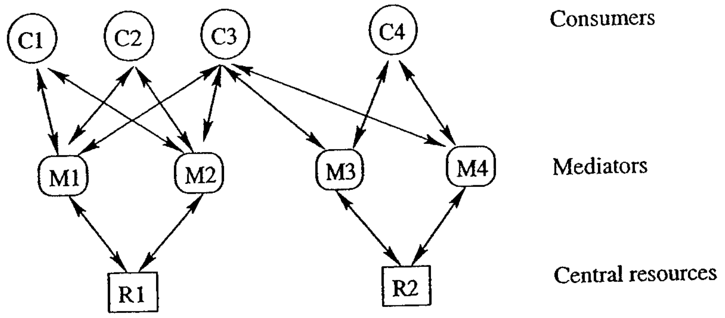


Figure 2: Overall architecture of the model

The only messages which are passed between agents (corresponding to the arcs in Figure 2) are as follows:

- A consumer can request a quantity of resource from a mediator.
- A mediator can credit a consumer with some quantity of resource.
- A mediator can inform its central resource that it has supplied a quantity of resource to a given consumer, along with the status of the resource account for that consumer at the time of supply.
- A central resource can give one of its mediators updated account information for a customer.

In the absence of consumers the central resource tends to increase (if the resource is money then this increase is due to returns on investments of the fund). We assume that this is proportional to the total amount in the fund. Consumers lose resource at what we assume is a fixed rate over time (if the resource is money this loss is through personal expenditure). If the funds remaining in a central resource or consumer reduce to zero that agent dies.

Consumers can employ different strategies for acquiring resources. We model two of these:

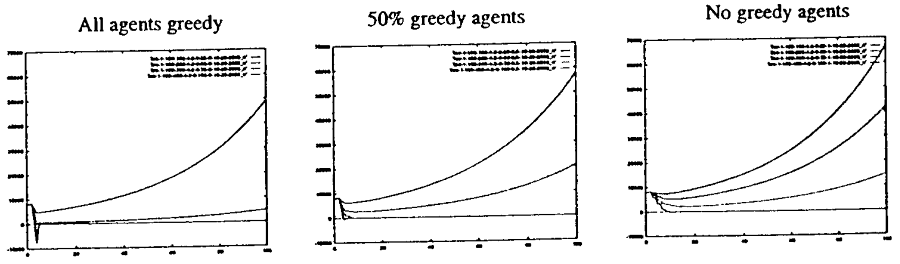
- A "careful" strategy in which a consumer applies to a single appropriate mediator each time it wants resource and waits until it has received credit for the resource before looking for more.
- A "greedy" strategy in which a consumer applies as often as possible for a given quantity of resource to all the mediators, with the aim of obtaining as much resource as it can as soon as it can.

This allows us to run the model with different settings for various parameters including the following (a range is given where we varied the parameter during analysis) :

Parameter	Setting
The initial number of resources.	4
The initial number of consumers.	100 - 400
The number of mediators per central resource.	2
The percentage of consumers allowed to interact with each central resource (the choice of individuals being a random subset of the total pool of consumers chosen according to this percentage).	25%
The percentage of greedy consumers in the total consumer population.	0-100%

2.3 Experimenting with the Model

A property of interest in this system is the ability of its central resources to be sustained under exploitation by different proportions of greedy consumers. We investigate this by fixing the initial level of central resource and running the model with different initial numbers of consumers. These runs are replicated with three different proportions of greedy consumers: 100%, 50% and 0%. The results are shown in Figure 3. The higher curves in each of the three diagrams are for runs with lower numbers of consumers, since these deplete the central resource less. The topmost curve of each graph, at 100 initial consumers, shows that at this population size the central resource recovers quickly from initial exploitation. The effects of having more greedy consumers appear most strongly at around 300 initial consumers, where the central resource diminishes to zero almost immediately when all or even half the agents are greedy, whereas it recovers after coming close to zero if no agents are greedy.


All agents greedy

50% greedy agents

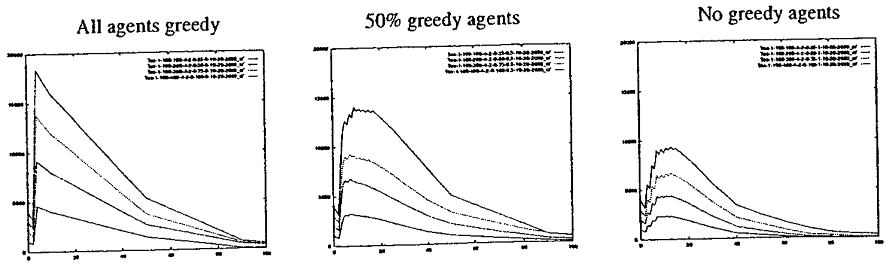
No greedy agents

Figure 3: Central resource variation in response to proportion of greedy consumers.

The consumers in this model have a basic strategy of claiming resource whenever they can (modulo the greedy or careful strategy) which means that they exhaust their allocations around midway through the simulation and then slowly die of starvation. This gives the pattern of sharp resource increase followed by slow tail-off shown in the graphs of Figure 4 (each of which gives the consumers' view of the corresponding simulation run in Figurefig:basicexptl). The peak is, unsurprisingly, most accentuated with greedy agents.

2.4 Counteracting a Threat

A surprising feature of the graph in Figurefig:basicexptl)involving greedy agents is that the central resource can be depleted to below zero. This is a side effect of having more than one mediators autonomously distributing the same resource because greedy consumers will attempt to collect resource from all mediators simultaneously. Each mediator, however, has its own separate knowledge of how much resource each consumer is allocated. Although



All agents greedy

50% greedy agents

No greedy agents

Figure 4: Consumer resource variation in response to agent mixture

this initially is consistent with the central resource it becomes inconsistent if more than one mediator simultaneously agrees to a release of resource for the same consumer, and this can cause more resource to be promised than the central resource has available. For instance if the resource allocated by R1 to C1 in Figure 2 is 10 units then M1 and M2 may simultaneously allow C1 to borrow 10 units. R1 now has to honour a commitment of 20 units although only 10 were intended. This phenomenon is known to occur in banking systems, where it is possible to withdraw more money than you have in your account by making multiple withdrawals through different mediators.

We would like to be able to counteract this threat but we cannot assume the easy solution of synchronising central resources and mediators each time a mediator is deciding whether to allocate resource to a consumer. An alternative is to create a new type of agent, call it an auditor, which at regular intervals asks a mediator and its central resource what they think is the unused allocation for a given consumer. If there is a discrepancy it broadcasts a warning message to all mediators who can then block further requests for resource from that consumer, ensuring that it starves to death. The arrangement is illustrated in Figure 5, where auditor A1 queries M1 and R1 about (say) C3 and finds a discrepancy so broadcasts a warning to M1, M2, M3 and M4.

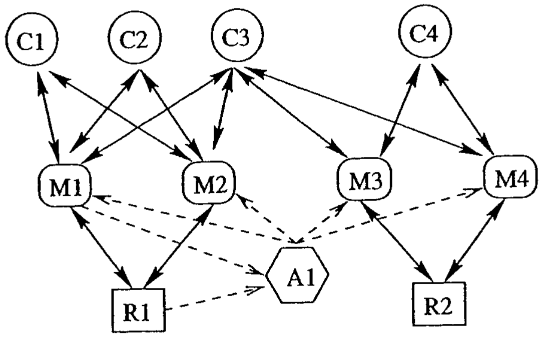


Figure 5: An auditor (A1) interacting with the resource system

To investigate whether this has a significant effect on the resilience of the resources against attack by greedy consumers simulations were run with 300 consumers, 50% of which were greedy, with first 0 auditors (as before) then 100 auditors; then 200 auditors. We can see the results of this in Figure 6. At 0 auditors the resource collapses to zero. With greater numbers of auditors it recovers quickly after an initial dip, as the warnings about greedy consumers exclude those consumers from the resources. We have not gone on to analyse whether there is an optimum number of auditors at which adding more gives lower increases in protection to the system but this seems likely to be the case.

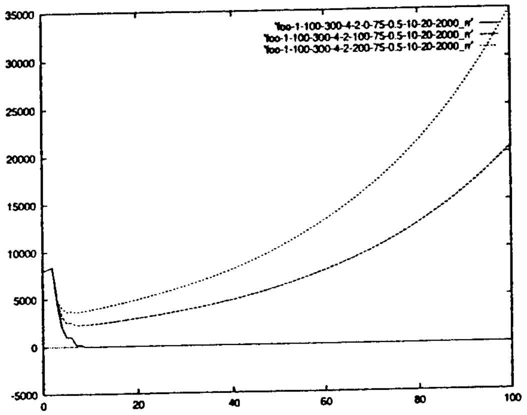


Figure 6: Shift in carrying capacity in response to auditing

Compositional Approach for Modeling and Verification of Component-Based Software Systems

Jeffrey J.P. Tsai and Eric Y.T. Juan
Department of Electrical Engineering and Computer Science
University of Illinois, Chicago
851 S. Morgan St., Chicago, IL 60607
Email: tsai@eecs.uic.edu

Abstract

With the rapid growth of networking and high-computing power, the demand of larger and more complex software systems has increased dramatically. To deal with the complexity in designing large-scale complex software systems, the concept of component-based software design has gained popularity recently. However, in pursuing a component-based approach there are obstacles to be overcome. One of them is the state-explosion problem in the formal verification of large-scale component-based systems. In this paper, we introduce a modeling technique and two condensation theories to model and verify component-based software systems. Our condensation theories are much weaker than current theories useful for the compositional verification. More significantly, our new condensation theories can eliminate the interleaved behaviors caused by asynchronously sending actions. Therefore, our technique provides a much more powerful means for the compositional verification of asynchronous processes. Our technique can efficiently analyze several state-based properties: deadlock state and reachable state. The experimental results show a significant improvement in the analysis of large-scale component-based systems.

1 INTRODUCTION

With the rapid growth of networking and high-computing power, the demand of larger and more complex software systems has increased dramatically. Examples include web-based systems, multimedia systems, telecommunication systems, intelligent agents systems, flight control systems, patience monitoring systems, robotics, virtual reality systems, and so on. However, the development of large-scale and complex software systems is much more difficult and error prone. This is due to the fact that techniques and tools for assuring the correctness and reliability of software systems lag far behind the increasing growth of size and complexity of software systems. The results are unreliable and poorly performing applications, delayed projects, and considerable cost overruns. In order to improve the usability and reliability of large-scale systems, the supporting techniques and development tools need to be greatly enhanced.

Formal verification is rapidly becoming accepted as a promising and automated method to verify the correctness of software systems. Despite many works in the area of formal verification, one of main bottlenecks so far is the state explosion problem. When the size of a software system increases linearly, the analysis complexity of the system could grow exponentially. The capability and performance of current techniques still can not efficiently verify typical large-scale software systems in practice.

A technique *"compositional verification"*, which is considered more suitable for analyzing well-defined subsystems, such as component-based systems, has been proposed by researchers to deal with the state-explosion problem of large-scale systems. However, current compositional verification techniques are efficient only for verifying event-based properties of synchronous processes. In this paper, we introduce a modeling technique and two condensation theories to model and verify component-based software systems. Our condensation theories are much weaker than current theories useful for the compositional verification. More significantly, our new condensation theories can eliminate the interleaved behaviors caused by asynchronously sending actions. Therefore, our technique provides a much more powerful means for the compositional verification of asynchronous processes. Our technique can efficiently analyze several state-based properties: deadlock state and reachable state. The experimental results show a significant improvement in the analysis of large-scale component-based systems.

2 THE MODEL

This section introduces a model, namely *multiset labeled transition systems* (MLTSs in the sequel). MLTSs are closely related to *labeled transition systems* (LTSs for short) which are intensively used as a state-space model in the family of process algebras. There are three distinguishing features between MLTSs and traditional LTSs. First, the label of a transition is a multiset of actions in MLTSs instead of one action in LTSs. Second, we make a clear distinction between synchronously communicating actions and asynchronously communicating actions. Third, the composition of traditional LTSs is for synchronous processes only, while the composition of both synchronous and asynchronous processes can be achieved in MLTSs. These features of MLTSs promise the development of a new mechanism for compositional verification. With the use of the new mechanism, the high analysis complexity of large-scale systems can significantly be reduced.

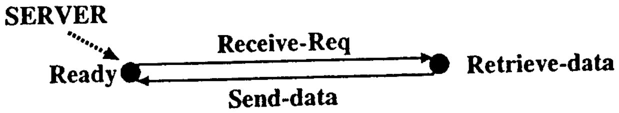


Figure 1: MLTS specification of a server.

The behavior of a process can be modeled by an MLTS. An MLTS consists of *states, transitions, actions,* and one *initial state*. In graphical representation of MLTSs, a *state* is denoted by a shaded circle, and a *transition* by a solid arrow labeled with *actions*. The *initial state* is pointed out by a dotted arrow.

A *state* in MLTSs could be interpreted as a condition. The states of an MLTS describe the possible conditions of a process. For instance, Figure 1 gives an MLTS which specifies a server. The condition of *SERVER* is either ready to accept a request (state *Ready*) or busy in retrieving data (state *Retrieve-data*). The state of an MLTS is changed by the execution of *actions*. Once *SERVER* performs the action of receiving request (*Receive-Req*), the condition of *SERVER* is changed from state *Ready* to state *Retrieve-data*. The condition of *SERVER* returns to state *Ready* after *SERVER* sends out data (action *Send-data*). An *initial state* is the condition at the beginning. The initial state of *SERVER* is *Ready*.

The relationship of *states* and *actions* is represented by *transitions*. A *transition* contains a *starting state*, one or more *actions*, and an *ending state*. For example, transition (*Ready, Receive-Req, Retrieve-data*) indicates that state *Retrieve-data* is *reachable* from state *Ready* by the execution of action *Receive-Req*. In the following, we first give the formal definition of MLTSs.

Definition 2.1 Multiset Labeled Transition Systems (MLTSs)

- A **multi-set** *MS* consists of a set D_{MS} and a mapping *MS*: $D_{MS} \rightarrow N$, where $N = \{1,2,3,...\}$ is the set of positive integers. *MS* is said to be a **multi-set** of a set *S* iff (if and only if) $D_{MS} \subseteq S$.
- A **MLTS** is a quadruple **(S, Σ_τ, T, s_{in})**, where

1) S is a set of **states**; s_{in} is the **initial state**;

2) Σ_τ, which is a set of **actions** (transition labels), comprises **invisible** (or **internal**) **action** (τ) and **communicating actions** Σ, where Σ consists of Σ_{syn} (**synchronously communicating actions**), Σ_{as} (**asynchronously sending actions**), and Σ_{ar} (**asynchronously receiving actions**); and

3) $\mathbf{T} \subseteq \mathbf{S} \times \mathbf{M}_{\Sigma_\tau} \times \mathbf{S}$ is a set of **transitions** such that $\forall$ *(s, m_s, s')* $\in$ **T**: m_s is a non-empty *multiset* of Σ_τ.

In MLTSs, a transition is labeled with a multiset of actions. A multiset consists of countable objects. This means that an action can have multiple instances in a transition label. We use *synchronous communication* and/or *asynchronous communication* as the primitive means of communication between processes. *Synchronously communicating actions* (Σ_{syn}) in MLTSs are used to specify unbuffered mode of synchronous communication which is usually referred to as handshaking or rendezvous communication. Synchronous communication between processes is performed through a simultaneous execution of actions which read from or write to a shared channel. In other words, actions which read from or write to a shared channel, have to take place at the same time.

The semantics of asynchronous interaction of MLTSs is essentially the same as asynchronous message passing used in NIL [5] and PLITS [2]. In asynchronous communication, a sending process is not blocked to wait for its communicating partners. Once a message is ready, the process is free to perform its asynchronously sending action. Messages which have not been received by receivers are stored in the buffers of channels. In MLTSs, the channel buffers of asynchronous communication have unbounded capability. An asynchronously receiving process, as in the case of synchronous communication, may be blocked in order to wait for messages. Note that channels for synchronous communication have no buffer for the storage of messages, since synchronous communication between processes has to take place at the same time.

3 DEADLOCK STATE AND REACHABLE STATE

In this section, we focus on the properties of deadlock states and reachable states. A state is said to be *reachable* in an MLTS if the state can be reached from the initial state via directed edges. Recall that a *state* in MLTSs could be interpreted as a condition. Thus, a *reachable state* can be considered as a possible condition that a process or system can reach.

A system is said to be *deadlocking* if the system has reached a condition (state) such that the system cannot do anything. In practice, a *deadlock state* reflects a failure or successful termination of a system. In order to distinguish failure from successful termination, we preserve the conditions of deadlocking systems, i.e., *deadlock states*. From the inspection of *deadlock states*, we can easily determine whether a deadlocking system fails or successfully terminates. In addition, a *deadlock state*, which provides detailed conditions of the whole system, is useful for debugging and modifying an improper system design. The following defines *reachable states* and *deadlock states* in MLTSs.

Definition 3.1 (Reachable States)

Let $(\mathbf{S}, \mathbf{\Sigma}_\tau, \mathbf{T}, \mathrm{s}_{in})$ be an MLTS. A state s is *reachable* in $(\mathbf{S}, \mathbf{\Sigma}_\tau, \mathbf{T}, \mathrm{s}_{in})$ iff

1) $s = \mathrm{s}_{in}$ or

2) $\exists\, (s_0, \mathrm{m}_1, \mathrm{s}_1) \ldots (\mathrm{s}_{n-1}, \mathrm{m}_n, \mathrm{s}_n)$ such that

i) $n \geq 1$, ii) $s_0 = s_{in}$, iii) $s_n = s$, and iv) $\forall\, 1 \leq j \leq n$: $(\mathrm{s}_{j-1}, \mathrm{m}_j, \mathrm{s}_j) \in \mathbf{T}$.

Definition 3.2 (Deadlock States)

Let $(\mathbf{S}, \mathbf{\Sigma}_\tau, \mathbf{T}, \mathrm{s}_{in})$ be an MLTS. A state s is a *deadlock state* of $(\mathbf{S}, \mathbf{\Sigma}_\tau, \mathbf{T}, \mathrm{s}_{in})$ iff

1) s is *reachable* in $(\mathrm{S}, \Sigma_\tau, \mathrm{T}, \mathrm{s}_{in})$ and 2) $\nexists\, (\mathrm{s}, \mathrm{m}, \mathrm{s}') \in \mathbf{T}$ (s has no outgoing transition).

4 CONDENSATION THEORIES FOR MLTSs

This section presents our newly derived *congruence theories* for the compositional verification of deadlock states and reachable states. We call our congruence theories *IOT-failure equivalence* and *IOT-state equivalence*. *IOT-failure equivalence* preserves the property of deadlock states while *IOT-state equivalence* preserves the property of reachable states. We will explain these two *congruence theories* using simple examples.

4.1 Paths and Input/Output-Traces (IO-Traces)

The computation of an MLTS can be described in terms of paths. A path is an alternating sequence of states and transitions in MLTSs. For example, MLTS P_7 in Figure 2 has a path

$$\sigma_7 = \{s_0, (s_0, Ch_1, s_1), s_1, (s_1, Ch_2\, !, s_2), s_2, (s_2, Ch_3\, !, s_3), s_3, (s_3, Ch_4, s_4), s_4\}.$$

For simplicity, we also write path σ_7 as

$$\{s_0, Ch_1, s_1, Ch_2\, !, s_2, Ch_3\, !, s_3, Ch_4, s_4\}.$$

Similarly, MLTS P_{7-C} in Figure 2 has a path

$$\sigma_{7-C} = \{s_0, (Ch_1, Ch_2\, !, Ch_3\, !), s_3, Ch_4, s_4\}.$$

Path σ_7 means that if the local condition of MLTS P_7 is state s_0, then P_7 is ready to sequentially execute actions $\{Ch_1, Ch_2\, !, Ch_3\, !, Ch_4\}$ and to sequentially reach states $\{s_1, s_2, s_3, s_4\}$. However, the execution along path σ_7 may fail due to some condition outside MLTS P_7, i.e., the environment condition of P_7. In other words, for the occurrence of a path to be successful, we need to consider the global condition which consists of a local condition and an environment condition.

We use *IO-traces* in order to deduce and compare the global conditions required for the occurrences of paths (actions). *IO-traces* are derived from paths by removing some details which are irrelevant to the success of paths' occurrences. Based on *IO-traces*, we have developed *IOT-failure equivalence* and *IOT-state equivalence* as presented in the following sections.

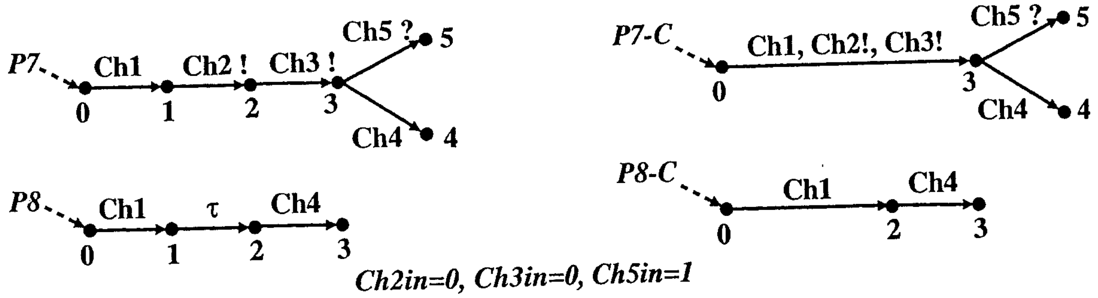


Figure 2: Example of paths and IO-traces.

An *IO-trace* consists of i) a starting state, ii) an ending state, and iii) a sequence of multisets of actions. Informally speaking, an *IO-trace* is derived from a path by

1) removing intermediate states,

2) replacing a transition with two ordered elements (multisets), i.e., i) its *environment pre-condition* and then ii) its *environment post-condition*,

3) removing empty multisets of actions, and

4) summing up adjacent multisets of *environment post-conditions*.

The *environment pre-condition* of a transition consists of *synchronously communicating actions* and *asynchronously receiving actions* which are labeled on the transition. The *environment post-condition* of a transition is represented by *asynchronously sending actions* labeled on the transition. The execution of *invisible actions* does not interact with the environment. Therefore, *invisible actions* are not included in *IO-traces*. Let us consider MLTSs in Figure 2 as an example. Assume that Ch_1 and Ch_4 are synchronous communication channels between processes P_7 and P_8. It is clear that the *environment pre-condition* of transition (s_0, Ch_1, s_1) is $\{Ch_1\}$; the *environment post-condition* of transition (s_0, Ch_1, s_1) is null. Now let us derive the *IO-trace* for the path $\sigma_7 = \{s_0, Ch_1, s_1, Ch_2\,!, s_2, Ch_3\,!, s_3, Ch_4, s_4\}$ in MLTS P_7 above. After 1) removing intermediate states s_1, s_2 and s_3, and 2) replacing a transition with its *environment pre-condition* and then its *environment post-condition*, we get

$$s_0, \{Ch_1\}, \{\,\}, \{\,\}, \{Ch_2\,!\}, \{\,\}, \{Ch_3\,!\}, \{Ch_4\}, \{\,\}, s_4.$$

After that, we remove empty multisets of actions and sum up adjacent multisets of *environment post-conditions* (or *asynchronously sending actions*). As a result, the *IO-trace* of path σ_7 is

$$IOT\sigma_7 = s_0, \{Ch_1\}, \{Ch_2\,!, Ch_3\,!\}, \{Ch_4\}, s_4.$$

Similarly, we can derive the *IO-trace* of path $\sigma_{7-C} = \{s_0, (Ch_1, Ch_2\,!, Ch_3\,!), s_3, Ch_4, s_4\}$ in MLTS P_{7-C} above as

$$IOT\sigma_{7-C} = s_0, \{Ch_1\}, \{Ch_2\,!, Ch_3\,!\}, \{Ch_4\}, s_4.$$

From paths σ_7 and σ_{7-C}, we can see that two different paths might have an identical *IO-trace*. *IO-traces* are useful for predicting the successful occurrences of paths (or actions). Based on *IO-traces*, *IOT-failure equivalence* and *IOT-state equivalence* will be presented in the following sections.

4.2 IO-Trace Failures (IOT-Failures)

In order to efficiently analyze a large-scale system, it is desirable to eliminate data which are irrelevant to the verification of interesting properties. Our *IOT-failure equivalence* is developed for the compositional verification of deadlock states. This means that *IOT-failure equivalent* MLTSs are interchangeable in the compositional verification of MLTSs without loss of any deadlock state. *IOT-failure equivalence* is very useful for reducing the size (complexity) of MLTSs when we focus on the property of deadlock states.

Informally speaking, an *IOT-failure* for an MLTS is a pair consisting of i) an *IO-trace t* starting from the initial state and ii) the set of *environment pre-conditions* for the outgoing transitions of the ending state of *t*. In addition, the ending state of the *IO-trace t* above should be *stable*.

A state s is said to be *stable* iff s does not have any out-transition whose *environment pre-condition* is empty; otherwise s is *non-stable*. In other words, only *stable states* are candidates for the construction of deadlock states, because a *non-stable* state has at least one out-transition which is guaranteed to be executable in any condition of the environment.

Let us consider MLTS P_7 in Figure 2 as an example. In MLTS P_7, both transitions (s_1, Ch_2 !, s_2) and (s_2, Ch_3 !, s_3) have an empty *environment pre-condition* because they execute neither *synchronously communicating action* nor *asynchronously receiving action*. Therefore, states s_1 and s_2 are *non-stable*. In contrast, states s_0, s_3, s_4, and s_5 are *stable*. These *stable states* are reached from the initial state s_0 via the following *IO-traces* respectively:

$IOT\sigma_0 = s_0, \phi, s_0,$
$IOT\sigma_3 = s_0, \{Ch_1\}, \{Ch_2\ !, Ch_3\ !\}, s_3,$
$IOT\sigma_4 = s_0, \{Ch_1\}, \{Ch_2\ !, Ch_3\ !\}, \{Ch_4\}, s_4,$ and
$IOT\sigma_5 = s_0, \{Ch_1\}, \{Ch_2\ !, Ch_3\ !\}, \{Ch_5\ ?\}, s_5.$

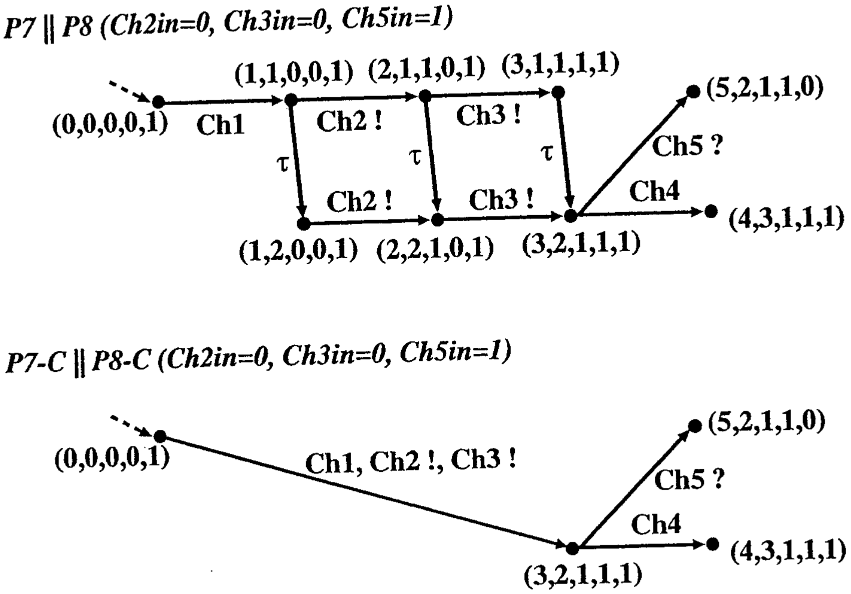
From these *stable states* and *IO-traces*, we get four *IOT-failures* in MLTS P_7:

$IOTF_0 = (IOT\sigma_0, \{Ch_1\}),$
$IOTF_3 = (IOT\sigma_3, \{\{Ch_4\}, \{Ch_5\ ?\}\}),$
$IOTF_4 = (IOT\sigma_4, \phi),$ and
$IOTF_5 = (IOT\sigma_5, \phi).$

IOT-failures $IOTF_4$ and $IOTF_5$ have an empty set of *environment pre-conditions* since states s_4 and s_5 have no out-transition. State s_3 has two out-transitions (s_3, Ch_4, s_4) and (s_3, Ch_5?, s_5). Transition (s_3, Ch_4, s_4) has an *environment pre-condition* $\{Ch_4\}$, and transition (s_3, Ch_5?, s_5) has an *environment pre-condition* $\{Ch_5\ ?\}$. Therefore, the set of *environment pre-conditions* of *IOT-failure* $IOTF_3$ is $\{\{Ch_4\}, \{Ch_5\ ?\}\}$.

Two MLTSs L_1 and L_2 are said to be *IOT-failure equivalent* iff L_1 and L_2 have the same set of *IOT-failures*. For instance, P_{7-C} in Figure 2 also has four *IOT-failures* $IOTF_0$, $IOTF_3$, $IOTF_4$, and $IOTF_5$. Thus, MLTSs P_7 and P_{7-C} in Figure 2 are *IOT-failure equivalent*. Similarly, MLTSs P_8 and P_{8-C} in Figure 2 are *IOT-failure equivalent* as well.

IOT-failure equivalence is a congruence in terms of deadlock states. Therefore, *IOT-failure equivalent* MLTSs are interchangeable in the compositional verification of MLTSs without loss of any deadlock state. For example, from the MLTSs in Figure 2, we can compose two MLTSs as shown in Figure 3. Without verifying these two MLTSs in Figure 3, we can guarantee that they have the same set of deadlock states, i.e., $s_{(5,2,1,1,0)}$ and $s_{(4,3,1,1,1)}$.



P7 || P8 (Ch2in=0, Ch3in=0, Ch5in=1)

P7-C || P8-C (Ch2in=0, Ch3in=0, Ch5in=1)

Figure 3: Deadlock-state equivalent MLTSs.

4.3 IO-Trace States (IOT-States)

IOT-state equivalence is developed for the compositional verification of reachable states. An *IO-trace state (IOT-state)* is an *IO-trace* starting from the initial state. Two MLTSs are said to be *IOT-state equivalent* if they have the same set of *IOT-states*. *IOT-state equivalence* is a *congruence* in terms of reachable states. This means that *IOT-state equivalent* MLTSs are interchangeable in the compositional verification of MLTSs without loss of any reachable state.

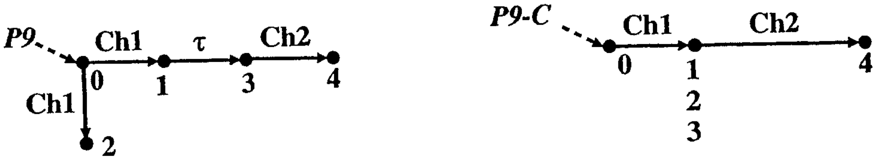


Figure 4: Example of *IOT-states* and *IOT-state equivalent* MLTSs.

As a simple example, let us consider MLTSs P_9 and P_{9-C} in Figure 4. Assume that Ch_1 and Ch_2 are synchronous communication channels. States s_1, s_2, and s_3 in MLTS P_9 are concisely represented by a macro state in MLTS P_{9-C}. We can see that MLTSs P_9 and P_{9-C} are *IOT-state equivalent* because they have the same set of *IOT-states*:

$IOTS_0 = s_0, \phi, s_0,$
$IOTS_1 = s_0, \{Ch_1\}, s_1,$
$IOTS_2 = s_0, \{Ch_1\}, s_2,$
$IOTS_3 = s_0, \{Ch_1\}, s_3,$ and
$IOTS_4 = s_0, \{Ch_1\}, \{Ch_2\}, s_4.$

5 CONCLUSION

This paper presents a new modeling technique and two new condensation theories to reduce the state explosion problem of asynchronous processes as well as synchronous processes in component-based software systems. Our condensation technique has reasonable complexity (polynomial in the numbers of states and transitions). From the experimental results, our technique promises a much more efficient analysis, especially for asynchronous processes in distributed systems. The condensation theories can be applied to Petri nets model too [6]. The current version of our technique focuses on the analysis of deadlock states and reachable states. Nevertheless, we believe that a more elaborated extension can be used to verify many other important safety and liveness properties of distributed systems, such as accessibility and event sequences. An extension of our work is currently under study.

6 ACKNOWLEDGMENTS

This research was supported in part by NSF and DARPA under Grant CCR-9633536.

References

[1] S. Brookes, C. Hoare, and A. Roscode, "A theory of communicating sequential processes," *ACM* 31, 3, pp. 560-599, 1984.

[2] J.A. Feldman, "A programming methodology for distributed computing (among other things)," *Communication ACM* 22, pp 353-368, 1979.

[3] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, N.J., 1985.

[4] R. Milner, "Operational and algebraic semantics of concurrent processes," *Handbook of theoretical computer science,* ed. J. van Leeuwen, Elsevier Science Publisher B.B., 1990.

[5] R.E. Strom and N. Halim, "A new programming methodology for long-lived software systems," *IBM J. Res. Devel.* 28, pp. 52-59, 1984.

[6] Y. T. Juan, J. J. P. Tsai, and T. Murata, "Compositional Verification of Concurrent Systems Using Petri-Nets-Based Condensation Rules," *ACM Transactions on Programming Languages and Systems*, Vol. 20, No. 5, pp. 917-979, Sept. 1998.

APPLYING MACHINE LEARNING ALGORITHMS IN SOFTWARE DEVELOPMENT

Du Zhang

Department of Computer Science
California State University
Sacramento, CA 95819-6021
zhangd@ecs.csus.edu

Abstract

Machine learning deals with the issue of how to build programs that improve their performance at some task through experience. Machine learning algorithms have proven to be of great practical value in a variety of application domains. They are particularly useful for (a) poorly understood problem domains where little knowledge exists for the humans to develop effective algorithms; (b) domains where there are large databases containing valuable implicit regularities to be discovered; or (c) domains where programs must adapt to changing conditions. Not surprisingly, the field of software engineering turns out to be a fertile ground where many software development tasks could be formulated as learning problems and approached in terms of learning algorithms. In this paper, we first take a look at the characteristics and applicability of some frequently utilized machine learning algorithms. We then provide formulations of some software development tasks using learning algorithms. Finally, a brief summary is given of the existing work.

Keywords: machine learning, software engineering, learning algorithms.

1. The Challenge

The challenge of modeling software system structures in a fastly moving scenario gives rise to a number of demanding situations. First situation is where software systems must dynamically adapt to changing conditions. The second one is where the domains involved may be poorly understood. And the last but not the least is one where there may be no knowledge (though there may be raw data available) to develop effective algorithmic solutions.

To answer the challenge, a number of approaches can be utilized [1,12]. One such approach is the *transformational programming*. Under the transformational programming, software is developed, modified, and maintained at specification level, and then automatically transformed into production-quality software through automatic program synthesis [5]. This software development paradigm will enable software engineering to become the discipline of capturing and automating currently undocumented domain and design knowledge [10]. Software engineers will deliver knowledge-based application generators rather than unmodifiable application programs.

In order to realize its full potential, there are tools and methodologies needed for the various tasks inherent to the transformational programming. In this paper, we take a look at how machine learning (ML) algorithms can be used to build tools for software development and maintenance tasks. The rest of the paper is organized as follows. Section 2 provides an overview of machine learning and frequently used learning algorithms. Some of the software development and maintenance tasks for which learning algorithms are applicable are given in Section 3. Formulations of those tasks in terms of the learning algorithms are discussed in Section 4. Section 5 describes some of the existing work. Finally in Section 6, we conclude the paper with remarks on future work.

2. Machine Learning Algorithms

Machine learning deals with the issue of how to build computer programs that improve their performance at some task through experience [11]. Machine learning algorithms have been utilized in: (1) data mining problems where large databases may contain valuable implicit regularities that can be discovered automatically; (2) poorly understood domains where humans might not have the knowledge needed to develop effective algorithms; and (3) domains where programs must dynamically adapt to changing conditions [11]. Learning a target function from training data involves many issues (function representation, how and when to generate the function, with what given input, how to evaluate the performance of generated function, and so forth). Figure 1 describes the dimensions of the target function learning.

Major types of learning include: concept learning (CL), decision trees (DT), artificial neural networks (ANN), Bayesian belief networks (BBN), reinforcement learning (RL), genetic algorithms (GA) and genetic programming (GP), instance-based learning (IBL), inductive logic programming (ILP), and analytical learning (AL). Table 1 summarizes the main properties of different types of learning.

Not surprisingly, machine learning methods can be (and some have already been) used in developing better tools or software products. Our preliminary study identifies the software development and maintenance tasks in the following areas to be appropriate for machine learning applications: requirement engineering (knowledge elicitation, prototyping); software reuse (application generators); testing and validation; maintenance (software understanding); project management (cost, effort, or defect prediction or estimation).

3. Software Engineering Tasks

Table 2 contains a list of software engineering tasks for which ML methods are applicable. Those tasks belong to different life-cycle processes of requirement specification, design, implementation, testing and maintenance. This list is by no means a complete one. It only serves as a harbinger of what may become a fertile ground for some exciting research on applying ML techniques in software development and maintenance.

One of the attractive aspects of ML techniques is the fact that they offer an invaluable complement to the existing repertoire of tools so as to make it easier to rise to the challenge of the aforementioned demanding situations.

4. Applying ML Algorithms to SE Tasks

In this section, we formulate the identified software development and maintenance tasks as learning problems and approach the tasks using machine learning algorithms.

Component reuse

Component retrieval from a software repository is an important issue in supporting software reuse. This task can be formulated into an instance-based learning problem as follows:

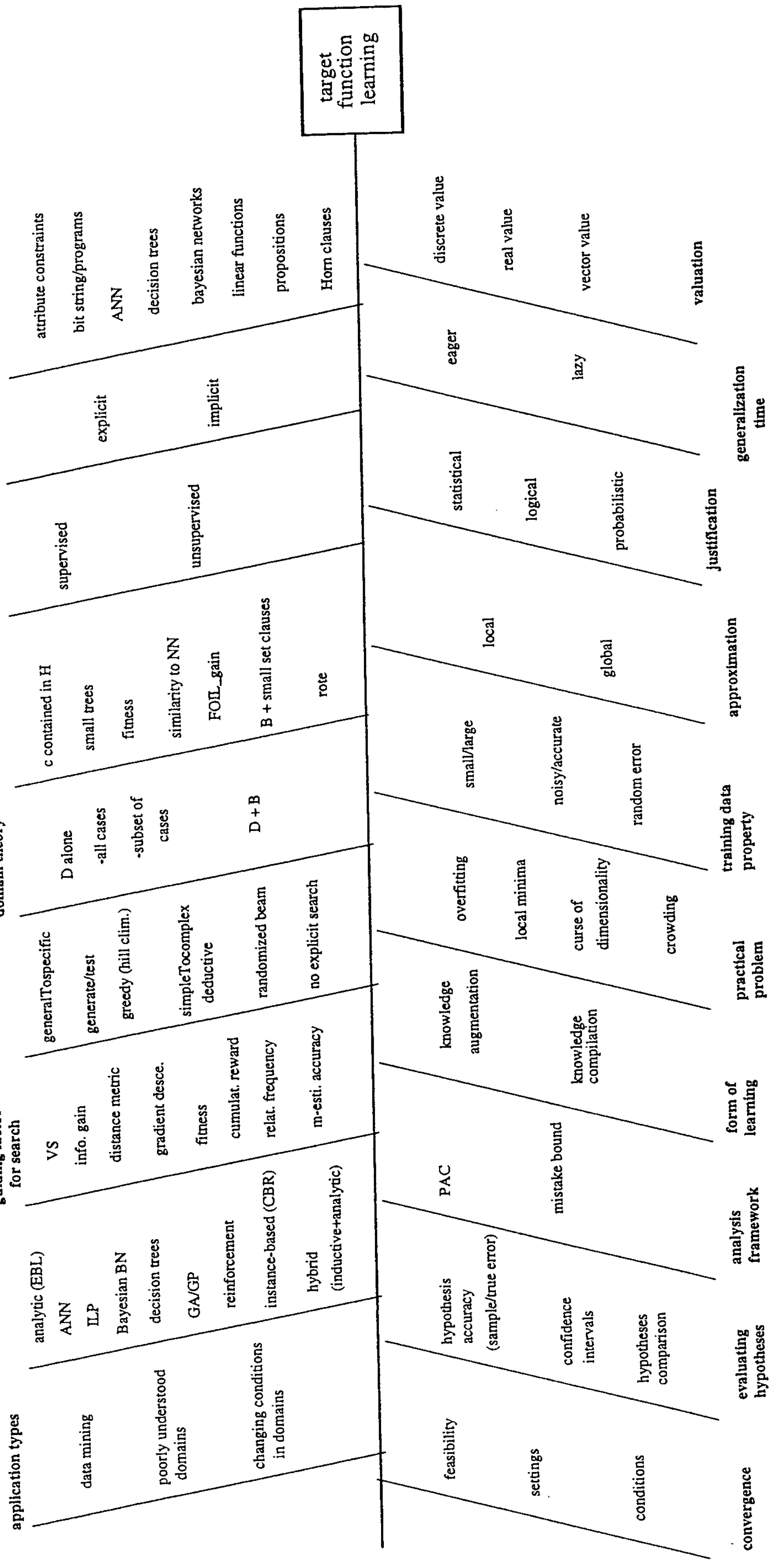


Figure 1. Dimensions of learning.

Table 1. Major types of learning methods[1].

Type	Target function	Target function generation[2]	Search	Inductive bias	Algorithm[3]
AL	Horn clauses	Eager, supervised, D + B	Deductive reasoning	B + set of Horn clauses	Prolog-EBG
ANN	ANN	Eager, supervised, D (global)	Gradient descent guided	Smooth interpolation between data points	Back-propagation
BBN	Bayesian network	Eager, supervised, D (global), explicit or implicit	Probabilistic, no explicit search	Minimum description length	MAP, BOC, Gibbs, NBC
CL	Conjunction of attribute constraints	Eager, supervised, D (global)	Version Space (VS) guided	$c \in H$	Candidate_ elimination
DT	Decision trees	Eager, supervised, D (global)	Information gain (entropy)	Preference for small trees	ID3, C4.5, Assistant
GA GP	Bit strings, program trees	Eager, unsupervised, no D	Hill climbing (simulated evolution)	Fitness-driven	Prototypical GA/GP algorithms
IBL	Not explicitly defined	Lazy, supervised, D (local)	Statistical reasoning	Similarity to NN	K-NN, LWR, CBR
ILP	If-then rules	Eager, supervised, D (global)	Statistical, general-to-specific	Rule accuracy, FOIL-gain, shorter clauses	SCA, FOIL, inverse resolution
RL	Control strategy π^*	Eager, unsupervised, no D	Through training episodes	Actions with max. Q value	Q, TD

[1] The classification here is based on materials in [11].

[2] The sets D and B refer to training data and domain theory, respectively.

[3] The algorithms listed are only representatives from different types of learning.

Table 2. SE tasks and applicable ML methods.

SE tasks	Applicable type(s) of learning
Requirement engineering	AL, BBN, LL, DT, ILP
Rapid prototyping	GP
Component reuse	IBL (CBR[4])
Cost/effort prediction	IBL (CBR), DT, BBN, ANN
Defect prediction	BBN
Test oracle generation	AL (EBL[5])
Test data adequacy	CL
Validation	AL
Reverse engineering	CL

1. Components in a software repository are represented as points in the n-dimensional Euclidean space (or cases in a case base).
2. Information in a component can be divided into *indexed* and *unindexed* information (attributes). Indexed information is used for retrieval purpose and unindexed information is used for contextual purpose. Because of the *curse of dimensionality* problem [11], the choice of indexed attributes must be judicious.
3. Queries to the repository for desirable components can be represented as constraints on indexable attributes.
4. Similarity measures for the nearest neighbors of the desirable component can be based on the standard Euclidean distance, distance-weighted measure, or symbolic measure.
5. The possible retrieval methods include: *K-Nearest Neighbor*, *inductive retrieval*, *Locally Weighted Regression*.
6. The adaptation of the retrieved component for the task at hand can be *structural* (applying adaptation rules directly to the retrieved component), or *derivational* (reusing adaptation rules that generated the original solution to produce a new solution).

Rapid prototyping

Rapid prototyping is an important tool for understanding and validating software requirements. In addition, software prototypes can be used for other purposes such as user training and system testing [18]. Different prototyping techniques have been developed for *evolutionary* and *throw-away* prototypings. The existing techniques can be augmented by including a machine learning approach, i.e., the use of genetic programming.

In GP, a computer program is often represented as a program tree where the internal nodes correspond to a set of functions used in the program and the external nodes (terminals) indicate variables and constants used as input to functions. For a given problem, GP starts with an initial population of randomly generated computer programs. The evolution process of generating a final computer program that solves the given problem hinges on some sort of fitness evaluation and probabilistically reproducing the next generation of the

[4] CBR stands for case-based reasoning.

[5] EBL refers to explanation-based learning.

program population through some genetic operations. Given a GP development environment such as the one in [8], the framework of a GP-based rapid prototyping process can be described as follows:

1. Define sets of functions and terminals to be used in the developed (prototype) systems.
2. Define a *fitness* function to be used in evaluating the worthiness of a generated program. Test data (input values and expected output) may be needed in assisting the evaluation.
3. Generate the initial program population.
4. Determine selection strategies for programs in the current generation to be included in the next generation population.
5. Decide how the genetic operations (*crossover* and *mutation*) are carried out during each generation and how often these operations are performed.
6. Specify the terminating criteria for the evolution process and the way of checking for termination.
7. Translate the returned program into a desired programming language format.

Requirement engineering

Requirement engineering refers to the process of establishing the services a system should provide and the constraints under which it must operate [18]. A requirement may be functional or non-functional. A functional requirement describes a system service or function, whereas a non-functional requirement represents a constraint imposed on the system. How to obtain functional requirements of a system is the focus here. The situation in which ML algorithms will be particularly useful is when there exist empirical data from the problem domain that describe how the system should react to certain inputs. Under this circumstance, functional requirements can be "learned" from the data through some learning algorithm.

1. Let X and C be the domain and the co-domain of a system function f to be learned. The data set D is defined as: $D = \{<x_i, c_k>| x_i \in X \wedge c_k \in C\}$.
2. The target functions f to be learned is such that $\forall x_i \in X$ and $\forall c_k \in C, f(x_i) = c_k$.
3. The learning methods applicable here have to be of *supervised* type. Depending on the nature of the data set D, different learning algorithms (in AL, BBN, CL, DT, ILP) can be utilized to capture (learn) a system's functional requirements.

Reverse engineering

Legacy systems are old systems that are critical to the operation of an organization which uses them and that must still be maintained. Most legacy systems were developed before software engineering techniques were widely used. Thus they may be poorly structured and their documentation may be either out-of-date or non-existent. In order to bring to bear the legacy system maintenance, the first task is to recover the design or specification of a legacy system from its source or executable code (hence, the term of reverse engineering, or program comprehension and understanding). Below we describe a framework for deriving functional specification of a legacy software system from its executable code.

1. Given the executable code p and its input data set X, and output set C, the training data set D is defined as: $D = \{< x_i, p(x_i)>| x_i \in X \wedge p(x_i) \in C\}$.
2. The process of deriving the functional specification f for p can be described as a learning problem in which f is learned through some ML algorithm such that
$$\forall x_i \in X\ [f(x_i) = p(x_i)].$$
3. Many supervised learning methods can be used here (e.g., CL).

Validation

Verification and validation are important checking processes to make sure that implemented software system conforms to its specification. To check a software implementation against its specification, we assume the availability of both a specification and an executable code. This checking process can be performed as an analytic learning task as follows:

1. Let X and C be the domain and co-domain of the implementation (executable code) p, which is defined as: p: X → C.
2. The training set D is defined as: $D = \{<x_i, p(x_i)> | x_i \in X\}$.
3. The specification for p is denoted as B, which corresponds to the domain theory in the analytic learning.
4. The validation checking is defined to be: p is valid if
$$\forall <x_i, p(x_i)> \in D \ [B \wedge x_i \vdash p(x_i)].$$
5. Explanation-based learning algorithms can be utilized to carry out the checking process.

Test oracle generation

Functional testing involves executing a program under test and examining the output from the program. An oracle is needed in functional testing in order to determine if the output from a program is correct. The oracle can be a human or a software one [13]. The approach we propose here allows a test oracle to be learned as a function from the specification and a small set of training data. The learned test oracle can then be used for the functional testing purpose.

1. Let X and C be the domain and co-domain of the program p to be tested. Let B be the specification for p.
2. Define a small training set D as: $D = \{<x_i, p(x_i)> | x_i \in X' \wedge X' \subset X \wedge p(x_i) \in C\}$.
3. Use the explanation-based learning (EBL) to generate a test oracle Θ (Θ: X → C) for p from B and D.
4. Use Θ for the functional testing: $\forall x_i \in X$ [output of p is correct if $p(x_i) = \Theta(x_i)$].

Test adequacy criteria

Software test data adequacy criteria are rules that determine if a software product has been adequately tested [21]. A test data adequacy criterion ζ is a function: ζ: $P \times S \times T \rightarrow$ {true, false} where P is a set of programs, S a set of specifications and T the class of test sets. $\zeta(p, s, t)$ = true means that t is adequate for testing program p against specification s according to criterion ζ. Since ζ is essentially a Boolean function, we can use a strategy such as CL to learn the test data adequacy criteria.

1. Define the instance space X as: $X = \{ <p_i, s_j, t_k> | p_i \in P \wedge s_j \in S \wedge t_k \in T\}$.
2. Define the training data set D as: $D = \{<x, \zeta(x)> | x \in X \wedge \zeta(x) \in V\}$, where V is defined as: V = {true, false}.
3. Use the concept of *version space* and the *candidate-elimination* algorithm in CL to learn the definition of ζ.

Software defect prediction

Software defect prediction is a very useful and important tool to gauge the likely delivered quality and maintenance effort before software systems are deployed [4]. Predicting defects requires a holistic model rather than a single-issue model that hinges on either size, or complexity, or testing metrics, or process quality data alone. It is argued in [4] that all these factors must be taken into consideration in order for the defect prediction to be successful.

Bayesian Belief Networks (BBN) prove to be a very useful approach to the software defect prediction problem. A BBN represents the *joint probability distribution* for a set of variables. This is accomplished by specifying (a) a directed acyclic graph (DAG) where nodes represent variables and arcs correspond to *conditional independence* assumptions (causal knowledge about the problem domain), and (b) a set of local conditional probability tables (one for each variable) [7, 11]. A BBN can be used to infer the probability distribution for a target variable (e.g., "Defects Detected"), which specifies the probability that the variable will take on each of its possible values (e.g., "very low", "low", "medium", "high", or "very high" for the variable "Defects Detected") given the observed values of the other variables. In general, a BBN can be used to compute the probability distribution for any subset of variables given the values or distributions for any subset of the remaining variables. When using a BBN for a decision support system such as software defect prediction, the steps below should be followed.

1. Identify variables in the BBN. Variables can be: (a) *hypothesis* variables for which the user would like to find out their probability distributions (hypothesis variable are either unobservable or too costly to observe), (b) *information* variables that can be observed, or (c) *mediating* variables that are introduced for certain purpose (help reflect independence properties, facilitate acquisition of conditional probabilities, and so forth). Variables should be defined to reflect the life-cycle activities (specification, design, implementation, and testing) and capture the multi-facet nature of software defects (perspectives from size, testing metrics and process quality). Variables are denoted as nodes in the DAG.
2. Define the proper causal relationships among variables. These relationships also should capture and reflect the causality exhibited in the software life-cycle processes. They will be represented as arcs in the corresponding DAG.
3. Acquire a probability distribution for each variable in the BBN. Theoretically well-founded probabilities, or frequencies, or subjective estimates can all be used in the BBN. The result is a set of conditional probability tables one for each variable. The full joint probability distribution for all the defect-centric variables is embodied in the DAG structure and the set of conditional probability tables.

Project effort (cost) prediction

How to estimate the cost for a software project is a very important issue in the software project management. Most of the existing work is based on algorithmic models of effort [17]. A viable alternative approach to the project effort prediction is instance-based learning. IBL yields very good performance for situations where an algorithmic model for the prediction is not possible. In the framework of IBL, the prediction process can be carried out as follows.

1. Introduce a set of features or attributes (e.g., number of interfaces, size of functional requirements, development tools and methods, and so forth) to characterize projects. The decision on the number of features has to be judicious, as this may become the cause of the *curse of dimensionality* problem that will affect the prediction accuracy.
2. Collect data on completed projects and store them as instances in the case base.
3. Define *similarity* or *distance* between instances in the case base according to the symbolic representations of instances (e.g., Euclidean distance in an n-dimensional space where n is the number of features used). To overcome the potential curse of

dimensionality problem, features may be weighed differently when calculating the distance (or similarity) between two instances.

4. Given a query for predicting the effort of a new project, use an algorithm such as *K-Nearest Neighbor*, or, *Locally Weighted Regression* to retrieve similar projects and use them as the basis for returning the prediction result.

5. Existing Work

Several areas in software development have already witnessed the use of machine learning methods. In this section, we take a look at some reported results. The list is definitely not a complete one. It only serves as an indication that people realize the potential of ML techniques and begin to reap the benefits from applying them in software development and maintenance.

Scenario-based requirement engineering

The work reported in [9] describes a formal method for supporting the process of inferring specifications of system goals and requirements inductively from interaction scenarios provided by stakeholders. The method is based on a learning algorithm that takes scenarios as examples and counter-examples (positive and negative scenarios) and generates goal specifications as temporal rules.

A related work in [6] presents a scenarios-based elicitation and validation assistant that helps requirements engineers acquire and maintain a specification consistent with scenarios provided. The system relies on explanation-based learning (EBL) to generalize scenarios to state and prove validation lemmas.

Software project effort estimation

Instance-based learning techniques are used in [17] for predicting the software project effort for new projects. The empirical results obtained (from nine different industrial data sets totaling 275 projects) indicate that cased-based reasoning offers a viable complement to the existing prediction and estimations techniques. A related CBR application in software effort estimation is given in [20].

Decision trees (DT) and artificial neural networks (ANN) are used in [19] to help predict software development effort. The results were competitive with conventional methods such as COCOMO and function points. The main advantage of DT and ANN based estimation systems is that they are adaptable and nonparametric.

The result reported in [3] indicates that the improved predictive performance can be obtained through the use of Bayesian analysis. Additional research on ML based software effort estimation can be found in [2,14,15,16].

Software defect prediction

Bayesian belief networks are used in [4] to predict software defects. Though the system reported is only a prototype, it shows the potential BBN has in incorporating multiple perspectives on defect prediction into a single, unified model.

Variables in the prototype BBN system [4] are chosen to represent the life-cycle processes of specification, design and implementation, and testing (Problem-Complexity, Design-Effort, Design-Size, Defects-Introduced, Testing-Effort, Defects-Detected, Defects-Density-At-Testing, Residual-Defect-Count, and Residual-Defect-Density). The proper causal relationships among those software life-cycle processes are then captured and reflected as arcs connecting the variables.

A tool is then used with regard to the BBN model in the following manner. For given facts about Design-Effort and Design-Size as input, the tool will use Bayesian inference to derive the probability distributions for Defects-Introduced, Defects-Detected and Defect-Density.

6. Concluding Remarks

In this paper, we show how ML algorithms can be used in tackling software engineering problems. ML algorithms not only can be used to build tools for software development and maintenance tasks, but also can be incorporated into software products to make them adaptive and self-configuring. A maturing software engineering discipline will definitely be able to benefit from the utility of ML techniques.

What lies ahead is the issue of realizing the promise and potential ML techniques have to offer in the circumstances as discussed in Section 4. In addition, expanding the frontier of ML application in software engineering is another direction worth pursuing.

References

1. B. Boehm, "Requirements that handle IKIWISI, COTS, and rapid change," *IEEE Computer*, Vol. 33, No. 7, July 2000, pp.99-102.
2. L. Briand, V. Basili and W. Thomas, "A pattern recognition approach for software engineering data analysis," *IEEE Trans. SE*, Vol. 18, No. 11, November 1992, pp. 931-942.
3. S. Chulani, B. Boehm and B. Steece, "Bayesian analysis of empirical software engineering cost models," *IEEE Trans. SE*, Vol. 25, No. 4, July 1999, pp. 573-583.
4. N. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Trans. SE*, Vol. 25, No. 5, Sept. 1999, pp. 675-689.
5. C. Green et al, "Report on a knowledge-based software assistant, In *Readings in Artificial Intelligence and Software Engineering*, eds. C. Rich and R.C. Waters, Morgan Kaufmann, 1986, pp.377-428.
6. R.J. Hall, "Systematic incremental validation of reactive systems via sound scenario generalization," *Automatic Software Eng*., Vol.2, pp.131-166, 1995.
7. F.V. Jensen, *An Introduction to Bayesian Networks*, Springer, 1996.
8. M. Kramer, and D. Zhang, "Gaps: a genetic programming system," *Proc. of IEEE International Conference on Computer Software and Applications* (COMPSAC 2000).
9. van Lamsweerde and L. Willemet, "Inferring declarative requirements specification from operational scenarios," *IEEE Trans. SE*, Vol. 24, No. 12, Dec. 1998, pp.1089-1114.
10. M. Lowry, "Software engineering in the twenty first century", *AI Magazine*, Vol.14, No.3, Fall 1992, pp.71-87.
11. T. Mitchell, *Machine Learning*, McGraw-Hill, 1997.
12. D. Parnas, "Designing software for ease of extension and contraction," *IEEE Trans. SE*, Vol. 5, No. 3, March 1979, pp. 128-137.
13. D. Peters and D. Parnas, "Using test oracles generated from program documentation," *IEEE Trans. SE*, Vol. 24, No. 3, March 1998, pp. 161-173.
14. A. Porter and R. Selby, "Empirically-guided software development using metric-based classification trees," *IEEE Software*, Vol. 7, March 1990, pp. 46-54.

15. A. Porter and R. Selby, "Evaluating techniques for generating metric-based classification trees," *J. Systems Software*, Vol. 12, July 1990, pp. 209-218.

16. R. Selby and A. Porter, "Learning from examples: generation and evaluation of decision trees for software resource analysis," *IEEE Trans. SE*, Vol. 14, 1988, pp.1743-1757.

17. M. Shepperd and C. Schofield, "Estimating software project effort using analogies", *IEEE Trans. SE*, Vol. 23, No.12, November 1997, pp. 736-743.

18. I. Sommerville, *Software Engineering*, Addison-Wesley, 1996.

19. K. Srinivasan and D. Fisher, "Machine learning approaches to estimating software development effort," *IEEE Trans. SE*, Vol. 21, No. 2, Feb. 1995, pp. 126-137.

20. S. Vicinanza, M.J. Prietulla, and T. Mukhopadhyay, "Case-based reasoning in software effort estimation," *Proc. 11th Int'l. Conf. On Information Systems*, 1990, pp.149-158.

21. H. Zhu, "A formal analysis of the subsume relation between software test adequacy criteria," *IEEE Trans. SE*, Vol.22, No.4, April 1996, pp.248-255.

AD NUMBER	DATE	DTIC ACCESSION NOTICE

1. REPORT IDENTIFYING INFORMATION

A. ORIGINATING AGENCY

University degli Studi di Genova ITALY

B. REPORT TITLE AND/OR NUMBER

2000 Monterey Workshop on Modelling Software

C. MONITOR REPORT NUMBER

8953 MA 03

D. PREPARED UNDER CONTRACT NUMBER

N68171-00-M-5681

2. DISTRIBUTION STATEMENT

APPROVED FOR PUBLIC RELEASE

DISTRIBUTION UNLIMITED

PROCEEDINGS

RE

1. ss

2. (

3. A

4. L

5. D

DT

1.

2.

20001026 083

S EDITIONS ARE OBSOLETE