Lecture Notes in Computer Science 1641

Dieter Hutter Werner Stephan
Paolo Traverso Markus Ullmann (Eds.)

Applied Formal Methods – FM-Trends 98

International Workshop on Current Trends
in Applied Formal Methods
Boppard, Germany, October 1998
Proceedings

Springer

Lecture Notes in Computer Science 1641

Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Dieter Hutter Werner Stephan
Paolo Traverso Markus Ullmann (Eds.)

Applied Formal Methods – FM-Trends 98

International Workshop on Current Trends
in Applied Formal Methods
Boppard, Germany, October 7-9, 1998
Proceedings

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Dieter Hutter
Werner Stephan
German Research Center for Artificial Intelligence (DFKI GmbH)
Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany
E-mail: {hutter/stephan}@dfki.de

Paolo Traverso
ITC-irst, Centro per la ricerca scientifica e tecnologica
via Sommarive, 18, I-38050 Povo (Trento), Italy
E-mail: leaf@itc.it

Markus Ullmann
Bundesamt für Sicherheit in der Informationstechnik (BSI)
Godesberger Allee 183, D-53175 Bonn, Germany
E-mail: ullmann@bsi.de

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Applied formal methods - FM trends 98 : proceedings / International Workshop on Current Trends in Applied Formal Methods, Boppard, Germany, October 7 - 9, 1998. Dieter Hutter ... (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 1999
(Lecture notes in computer science ; Vol. 1641)
ISBN 3-540-66462-9

CR Subject Classification (1998): F.3, D.2, D.1, D.3, I.2, F.4.1

ISSN 0302-9743
ISBN 3-540-66462-9 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

© Springer-Verlag Berlin Heidelberg 1999
Printed in Germany

Typesetting: Camera-ready by author
SPIN: 10703472 06/3142 – 5 4 3 2 1 0 Printed on acid-free paper

Preface

This volume contains the contributions presented at the *International Workshop on Current Trends in Applied Formal Methods* organized October 7-9, 1998, in Boppard, Germany.

The main objective of the workshop was to draw a map of the key issues facing the practical application of formal methods in industry. This appears to be particularly timely with safety and security issues becoming a real obstacle to industrial software and hardware development. As a consequence, almost all major companies have now set up departments or groups to work with formal methods and many European countries face a severe labour shortage in this new field. Tony Hoare's prediction of the art of software (and hardware) development becoming a proper engineering science with its own body of tools and techniques is now becoming a reality.

So the focus of this application oriented workshop was not so much on special academic topics but rather on the many practical aspects of this emerging new technology: verification and validation, and tool support and integration into the software life-cycle. By evaluating the state of the art with respect to industrial applications a discussion emerged among scientists, practising engineers, and members of regulatory and funding agencies about future needs and developments. This discussion lead to roadmaps with respect to the future of this field, to tool support, and potential application areas and promising market segments. The contributions of the participants from industry as well as from the respective national security bureaus were particularly valuable and highly appreciated.

The workshop program included four invited talks, 16 talks selected by the program committee, and six talks given during the application day. Apart from the regular papers, the proceedings contain two invited contributions by Egon Börger and Manfred Broy, three application papers, and eight tool papers.

Egon Börger gives a new comprehensive introduction to Abstract State Machines describing their use in the different design stages of software systems. Manfred Broy and Oscar Slotosch present a process model for the integration of formal methods into conventional software engineering. They discuss the mediating role of user-oriented description techniques. Contributions presented at the application day cover industrial applications of model-checking (an excellent survey of current industrial practice was presented in the invited talk by Wolfram Büttner, SIEMENS), hardware-design (paper by Masahiro Fujita, Sree P. Rajan, and Alan Hu), safety critical control systems (paper by Meine van der Meulen and Tim Clement), and security engineering (paper by Frank Koob, Markus Ullmann, and Stefan Wittmann, BSI). The tool papers provide a collection of the major current systems.

October 1998 Jörg H. Siekmann

Welcome Note

I welcome you cordially on behalf of the Federal Minister of the Interior and the Federal IT Security Agency at this international workshop, and I extend a special welcome to the many foreign guests and the invited speakers of the conference.

To look into the future and identify technological trends is a challenge that raises many questions. If one considers IT development in general, one comes to the conclusion that it is not unusual to talk about and handle technologies that have existed for fewer than 5 years now. Java and the World Wide Web are two illustrative examples. Their development and distribution is taking place at a breathtaking speed. I do not think that anybody would have predicted that these phenomena would take on such dimensions.

At the moment, technical systems in a wide range of fields are becoming ever more complex. Computer failures (breaking of access codes, T-online) and accidents (plane crashes, train disasters) arouse a latent feeling in society that technical systems are no longer fully controllable. The impression arises that against this background security will increasingly have to be discussed in its entirety, and that new promising procedures will have to be applied. We expect that it will be possible to achieve major improvements in terms of quality and security through *Formal Methods* (FM) in the field of software and hardware systems.

Yet new technologies alone are not enough. Those who make the laws must take action, too, in order to ensure that the technologies promising a higher level of security are used appropriately, and are not sacrificed for short-term economic profit. A high-level information technical security infrastructure is essential to ensure confidence in the flow of information on the data highways and acceptance of new technologies in politics, the economy, and society. The Digital Signature Act and Ordinance as well as the related detailed technical specifications have now created a basis in Germany for the development of a uniform security infrastructure. The high certification level of the components to be used means that the use of FM for digital signature components that comply with the law is mandatory.

The aim of the workshop, which starts today here at the Federal Academy, is, on the one hand, to provide information about current developments of IT, and to discuss and possibly orientate them towards the requirements of industry. On the other hand, you will exchange experience regarding the use of FM during the Application Day and thus encourage industrial companies to use FM in order to make a contribution to increasing security and safety.

I wish the workshop the best of success, which I combine with the hope that the workshop will give a clear signal for a wider applicability of formal methods.

October 1998

MinR Norbert Vogt
German Federal Ministry of the Interior

Workshop Organization

The *International Workshop on Current Trends in Applied formal Methods* (FM-Trends 98) was organized jointly by the DFKI GmbH (Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken), the BSI (Bundesamt für Sicherheit in der Informationstechnik, Bonn), the JRC (Joint Research Centre of the European Commission, Ispra) and the IRST (Istituto per la Ricerca Scientifica e Tecnologica, Trento).

Workshop Committee

F. Giunchiglia(Trento)
D. Hutter (Saarbrücken)
F. Koob (Bonn)
J.H. Siekmann (Saarbrücken)
W. Stephan (Saarbrücken)
P. Traverso (Trento)
M. Ullmann (Bonn)
H.P. Wagner (Bonn)
M. Wilikens (Ispra)

Advisory Board

E. Astesiano, (Genova)
K.R. Apt, (Amsterdam)
J. Bergstra, (Amsterdam)
V. Berzins, (Monterey)
D. Bjorner, (Lyngby, Denmark)
R. Bloomfield, (Adelard)
J. Bowen, (Berkshire)
B. Boyer, (Texas)
M. Broy, (Munich)
A. Bundy, (Edinburgh)
E. Clarke, (CMU, USA)
W. Damm, (Oldenburg)
J.W. de Bakker, (Amsterdam)
W.P. de Roever, (Kiel)
H.D. Ehrich, (Braunschweig)
E.A. Emerson, (Texas)
H. Ganzinger, (Saarbrücken)
M.C. Gaudel, (Paris)
J. Goguen, (Monterey)
D. Gries, (Ithaca, New York)
Y. Gurevich, (Michigan)
D. Harel, (Rehovot, Israel)
T. Henzinger, (Berkeley)
M. Hinchey, (New Jersey)
C.A.R. Hoare, (Oxford)
D. Howe, (Bell Labs, USA)
N. Jones, (Copenhagen)
D. Kapur, (New York)
H. Kirchner, (Rocquencourt, France)
H. Langmaack, (Kiel)
T. Maibaum, (London)
U. Martin, (St. Andrews)
U. Montanari, (Pisa)
J S. Moore, (Texas)
T. Nipkow, (Munich)
D. Parnas, (Hamilton)
L. Paulson, (Cambridge)
A. Ravn, (Lyngby, Denmark)
A. W. Roscoe, (Oxford)
J. Sifakis, (Paris)

External Referees

Massimo Benerecetti
Piergiorgio Bertoli
Alessandro Cimatti
Marco Daniele
Dominique Mery
Olaf Müller
Marco Roveri
Roberto Sebastiani
Oscar Slotosch
Adolfo Villafiorita

Acknowledgements

All submitted papers were refereed by advisory board members, workshop organizers, and external referees. This workshop would not have been possible without their voluntary and dedicated work.

Table of Contents

Invited Papers

High Level System Design and Analysis Using Abstract State Machines ... 1
Egon Börger

Enriching the Software Development Process by Formal Methods 44
Manfred Broy, Oscar Slotosch

Regular Papers

Formal Program Development in Geometric Modeling 62
François Puitg, Jean-François Dufourd

Design of Distributed Multimedia Applications (DAMD) 77
Wanderley Lopes de Souza, Paulo Nazareno Maia Sampaio, Jean-Marie Farines, Roberto Milton Scheffel, Maria Janilce B. Almeida, Luciano Paschoal Gaspary, Lissandro Zambenodetti Granville, Roberto Willrich, Murilo S. de Camargo, Marcelo Domingos

Structured Formal Verification of a Fragment of the IBM S/390 Clock Chip 92
Alfons Geser, Wolfgang Küchlin

Automated Test Set Generation for Statecharts 107
K. Bogdanov, M. Holcombe, H. Singh

Rigorous Compiler Implementation Correctness: How to Prove the Real Thing Correct 122
Wolfgang Goerigk, Ulrich Hoffmann

Translation Validation: From DC+ to C 137
A. Pnueli, O. Shtrichman, M. Siegel

A Practical Hierarchical Design by Timed Simulation Relations for Real-Time Systems 151
Satoshi Yamane

A Lightweight Approach to Formal Methods 168
Sten Agerholm, Peter Gorm Larsen

An Open Environment for the Integration of Heterogeneous Modelling Techniques and Tools 184
Robert Büssow, Wolfgang Grieskamp, Winfried Heicking, Stephan Herrmann

Integrating Domain Specific Language Design in the Software Life Cycle . 196
Philipp W. Kutter, Daniel Schweizer, Lothar Thiele

Flexible and Reliable Process Model Properties: An Integrated Approach . 213
Tiziana Margaria, Volker Gruhn

A Symbolic Model Checker for ACTL 228
A. Fantechi, S. Gnesi, F. Mazzanti, R. Pugliese, E. Tronci

Critical Systems Validation and Verification with CSP and FDR 243
Michael Goldsmith, Irfan Zakiuddin

UniForM Perspectives for Formal Methods 251
Bernd Krieg-Brückner

The UniForM WorkBench - A Higher Order Tool Integration Framework . 266
Einar W. Karlsen

Application Papers

Two Real Formal Verification Experiences: ATM Switch Chip and Parallel Cache Protocol 281
Masahiro Fujita, Sree P. Rajan, Alan Hu

Formal Methods in the Specification of the Emergency Closing System of the Eastern Scheldt Storm Surge Barrier 296
Meine van der Meulen, Tim Clement

The New Topicality of Using Formal Models of Security Policy within the Security Engineering Process 302
Frank Koob, Markus Ullmann, Stefan Wittmann

Tool Papers

Towards Comprehensive Tool Support for Abstract State Machines: The ASM Workbench Tool Environment and Architecture 311
Giuseppe Del Castillo

The IFAD VDM Tools: Lightweight Formal Methods 326
Sten Agerholm, Peter Gorm Larsen

KIV 3.0 for Provably Correct Systems 330
Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel

PVS: An Experience Report 338
S. Owre, J. M. Rushby, N. Shankar, and D. W. J. Stringer-Calvert

QUEST: Overview over the Project 346
Oscar Slotosch

VSE: Controlling the Complexity in Formal Software Developments 351
Dieter Hutter, Heiko Mantel, Georg Rock, Werner Stephan, Andreas Wolpers, Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel

The wHOLe System 359
Mark E. Woodcock

Z/EVES Version 1.5: An Overview 367
ORA Canada

Author Index 377

High Level System Design and Analysis Using Abstract State Machines

Egon Börger

Università di Pisa, Dipartimento di Informatica, Corso Italia 40,
I-56125 Pisa, Italy, boerger@di.unipi.it

Abstract. We provide an introduction to a practical method for rigorous system development which has been used successfully, under industrial constraints, for design and analysis of complex hardware/software systems. The method allows one to start system development with a trustworthy high level system specification and to link such a "ground model" in a well documented and inspectable way through intermediate design steps to its implementation. The method enhances traditional operational modelling and analysis techniques by incorporating the most general abstraction, decomposition and refinement mechanisms which have become available through Gurevich's Abstract State Machines. Through its versatility the ASM approach is non-monolithic and integratable at any development level into current design and analysis environments. We also collect experimental evidence for the ASM thesis, a generalization of Turing's thesis.

1 Introduction

In [21] the methodological and epistemological reasons are explained why Gurevich's concept of Abstract State Machines (ASMs) allowed us to develop a mathematically well founded approach to software and hardware design and analysis which is nevertheless practical, improving system design practice by a piece of useful theory. Due to the progress the method and its industrial applications have made since then, some of the predictions and hopes expressed in [21] have become reality. It is therefore with pleasure that I accept the invitation to write a general introduction to the approach.

The first section contains the definition of ASMs and an illustration of the main characteristics of the method, namely the most general *abstraction mechanism* it offers together with the corresponding *refinement* technique for vertical structuring of systems and the *decomposition* technique for horizontal structuring. We define the *function classification* which the applications have led us to introduce into ASMs as practical support for abstraction, refinement and (de)composition in concise high-level modelling of large systems. We show how ASMs combine abstraction with rigour—circumventing the omnipresent "formal system straitjacket"—and how their versatility permits one to integrate

[0] Part of the material appearing here has been presented to the ASM Workshop held in Magdeburg, September 21-22, 1998, as part of the Annual GI-Meeting.

them into existing development environments and to apply to them established analysis (validation and verification) techniques. Since this holds also at high hardware or software design levels, ASMs help to avoid and to detect errors in the early stages of development, where reasoning, verification, simulation and testing are less expensive than in the later stages. In the second section we show that ASMs encompass in a natural way other well known approaches to system design and collect experimental evidence for Gurevich's ASM Thesis [76] which strengthens the Church-Turing thesis. Concluding we point to some directions for future research and applications.

2 Characteristics of the ASM Method

Gurevich [71–74] discovered the notion of ASM in an attempt to sharpen the Church-Turing thesis by considerations from complexity theory (bounded resources) and abstract data structures. I tried out ASMs in an attempt to model in a rigorous but transparent way the dynamic semantics of the programming language PROLOG (see [18,19]). This led to the ISO standard definition of the entire language [22,39,23] and became the starting point for the definition of a proven to be correct scheme for the implementation of Prolog on the Warren Abstract Machine [40]. Through this work and its various extensions [15,16,41, 34,37,42] we realized that on the basis of the most general abstraction mechanism inherent in ASMs, one can build a practical design method which can solve the fundamental triple problem of system development, namely

- to elaborate the informally presented requirements of the desired system turning them into a satisfactory *ground model*, i.e. a functionally complete but abstract description of sufficient but not more than necessary rigor which a) can be read and understood by and justified to the customer as solving his problem, b) defines every system feature as far as this is semantically relevant for the work the user expects the system to achieve, c) contains only what the logic of the problem requires for the system behavior, i.e. does not rely upon any further design decision belonging to the system implementation,
- to implement the ground model in a reliable manner by *refining* it step by step, in a controlled and well documented way, *through a hierarchy of intermediate models* each of which reflects some major design decision(s),
- to *structure the system horizontally* (from the ground model specification through the entire design process) by building it from components with abstract but sufficiently detailed definitions of the behavior of each single component and of the interaction of the components through interfaces.

This section contains the definition of ASMs and an illustration of this method through modelling examples taken from the literature. (For a detailed analysis of the ASM literature see [33] and [2].)

2.1 Abstraction Mechanism and Definition of ASMs

It has become a common place among computer science theoreticians and practitioners that most general abstraction principles are needed for coping with

the complexity of present day hardware and software systems (for an illuminating early survey see [119]). The disagreement and the problems with concrete realizations of this maxime begin with the definition of abstraction.

On the one hand the available theories for composing abstract data or actions (e.g. the sophisticated algebraic specification [133] and action semantics theories [99]) are not practical for descriptions of complex dynamic real-life systems. They make it difficult to follow the guidance of a given problem for adequately choosing and formulating the data and the actions, either in combination or independently of each other, at the abstraction level which is appropriate for the application under study. On the other hand even the most advanced among the currently available practical approaches suffer from serious limitations. They tend to restrict the abstractions provided for data and operations (typically offering basic manipulations of sets, sequences, mappings, etc., as for example in the VDM approach [60]) or to impose particular forms for defining action abstractions (like the syntactic definition of atomic actions in the B-Method [3] by means of substitutions). Such restrictions typically reflect design decisions which have been made a priori for a fixed (conceptual or tool based) framework for refining specifications to executable implementations. By letting such implementation concerns creep into the fundamentals of specification, a heavy tribute is paid to the particular solution, in the underlying framework or tool environment, of the representation problem for abstract data or actions.

Abstract States. Using ASMs one can (and is advised to) separate the high level specification concerns from concerns related to further design, without creating a gap between specification and design. ASMs allow us to produce rigorous high-level definitions which reflect in a direct (i.e. coding free) way the application domain knowledge and the connotations of the problem as it appears to the application domain expert. One can tune the formulation of the data and operations in terms of the concepts (objects and actions) which appear in the problem domain, freed from having to think about how to represent them through a priori fixed schemes for defining data types and actions. Technically this *freedom of abstraction*—meaning freedom of listening to the subject, thinking semantically in terms of objects and effects of actions and not in terms of their representations or schemes of definition—is realized by the concept of machine state and of state transforming machine instructions.

An ASM state is not an indivisible entity like the control state of a finite automaton, it is not a mere set (a collection of values) and also not a function (an association of values to variables); it is an arbitrarily complex (or simple) structure in the general sense the term is used in mathematics and has been defined by Tarski [127], i.e. a collection of domains (sets, also called universes, each of which stands for a particular type of objects) with arbitrary functions and relations defined on them. What a machine "knows" about the elements of its domains, and about the functions and predicates it uses for manipulating them, depends on what is there in the particular application for which the machine has been defined. A universe can be completely abstract, meaning that

no restriction is imposed on (and therefore no knowledge is available yet about) its elements, in particular not on their possible representation in our language or system. If the domain elements are assumed to have certain properties or to be in certain relations with other objects or to be subject to certain manipulations, we have to formulate these by corresponding operations, predicates or by conditions (integrity constraints) which the elements are required to satisfy.

As will become clear below, the ASM approach does not prescribe any particular notation to formulate constraints on machine states, whether verbally or formally, in programming, algorithmic, mathematical or any other rigorous or if necessary formal language. This has a pragmatically important effect on the integration potential of the approach (see section 2.5). Obviously the overall rigor of a description depends among other things on the degree of precision which is chosen for expressing such integrity constraints.

Before we illustrate the use of abstract machine states by four examples, let us say a word to avoid a possible misunderstanding which may come from the different use of the term "function" in mathematics and in programming. What we refer to is the mathematical meaning of the word function as a set of $n+1$-tuples with a uniqueness property (the functional dependence of the $n+1$-th element of a tuple from the first n elements, its arguments). Such a function corresponds to an n-dimensional array of programming. In particular nullary functions are the usual "programming variables", in the object-oriented spirit "class variables" which do not depend on the class members, whereas a monadic function $f : A \to B$ can be thought of as "instance variable" belonging to the class A, to be instantiated for each class member.

Computers hosting PVM Processes. Imagine you want to analyse the PVM [63] communication mechanism between application programs which run concurrently on physically interconnected heterogeneous machines of various architectures (serial, parallel, vector computers). Without going into irrelevant representation details we can describe the role of the constituting member computers by declaring them to be elements of a dynamically changing set *HOST* which comes with three informations. A 0-ary function (class variable)

$$master : HOST$$

describes a designated host machine on which PVM is started and which plays a special role for maintaining the control over the dynamically changing PVM configuration. Another function

$$arch : HOST \to ARCH$$

defines for each host machine its architecture which belongs to an abstract domain *ARCH* (definable as the set of possible architectures to be used with PVM 3 as listed in [63]). A third function

$$daemon : HOST \to DAEMON$$

dynamically provides each host machine with an abstract daemon process which acts as a PVM supervisor for the local management of application programs

(which are enrolled as processes into PVM) or for the communication between such processes (which may reside on different host machines). For the initialization a 0-ary function

$$demiurge : DAEMON$$

provides a distinguished daemon who resides on the master host (expressed by $daemon(master) = demiurge$) and is responsible in particular for creation and deletion of hosts and maintenance of the corresponding communication structure. These abstract domains and functions are the basis for the ASM model of PVM in [28, 29]. No further information is needed about what hosts, daemons, architectures and processes "are" or how they could be "represented".

Avoiding premature object representations. Working with abstract domains and functions allows us to avoid premature object representation decisions. In a practical modelling approach these *should* be avoided[1] because they typically enforce dealing with mere formalism (or performance) problems which carry no conceptual content for the problem under investigation and deviate the attention from it. This can be illustrated by the modelling example presented in the chapter on "Constructing a Model" in [60] "to provide some initial guidance on how one can start developing formal models using VDM-SL". A call-out mechanism has to be designed for a chemical plant alarm system where experts with different kinds of qualification must be called for coping with different kinds of alarm. The VDM-SL model represents experts by a record type with an expert identifier and a qualification field. It is explained (op.cit.p.18) that "the representation of ExpertID is important in the final implementation, but in this model none of the functions need to be concerned with its final representation. All that is required at this level of abstraction is for the identifiers to be values that can be compared for equality". As a consequence the authors propose *ExpertId=token* as "the completed type definition in VDM-SL", adding that "In VDM-SL, the special type representation called *token* is used to indicate that we are not concerned with the detailed representation of values, and that we only require a type which consists of some collection of values." All this does not prevent the authors from having to introduce, see op.cit. page 24, yet another condition in order "to ensure that ... one could not erroneously have two experts with different qualifications having the same expert identification". In an ASM model for the alarm system these complications disappear by treating experts abstractly as elements of a set *EXPERT* which comes with a function

$$quali : EXPERT \to QUALIFICATION$$

associating qualifications to experts. The doubling of experts and of expert identifiers (by the way also the introduction of a special type representation *token*) are avoided and the expert identification problem (which does interest our customer) is solved free of charge by the underlying classical logic of equality applied

[1] "Data in the first instance represent abstractions from real phenomena and are preferably formulated as abstract structures not necessarily realized in common programming languages."[135, p.10]

to elements of sets. Certainly one has to guarantee an appropriate internal representation of experts in the implementation (why not by records), but this is a problem which does not concern the customer any more and which is solved when the specification is refined to excutable code (see the discussion of refinement techniques in section 2.3).

Annotated program syntax trees. Typical usage of abstract domains and functions can be illustrated also through the ASM modelling of the dynamic semantics of various real-life programming languages like PROLOG [39], OCCAM [27], VHDL [30,31], JAVA [43] and many others [33]. Here the problem consists in expressing the static language concepts (objects/operations) and the dynamic program actions (effect of basic instructions) faithfully, the way they appear in the standard or language reference manual, without introducing any extraneous encoding due to an a priori fixed abstraction level and avoiding in particular any a priori imposed static representation of actions-in-time. In [27] we expressed the canonical (syntactically defined) part of the sequential control by a graph[2], i.e. a statically defined successor function on an abstract domain *NODE*. The nodes are decorated by a function

$$cmd : NODE \rightarrow INSTRUCTION$$

with atomic commands to be executed. This circumvents the typical encodings with terms (generated by the underlying context free grammar) or with continuations (typically appearing in denotational descriptions) or tables and the like and allows one to formulate the meaning of instructions locally (providing a modular definition of semantics) and such that it directly reflects the relevant language design decisions. In this way the ASM model for Occam [27] lets the dynamics of the distributed language features stand out explicitly in rules which govern the local execution of program instructions by daemons. The daemons are elements of an abstract dynamic set *DAEMON*; they are created and deleted and, carrying along their private environment, walk through the graph (updating a positioning function *loc*). They move in a truly concurrent way, independently from each other (unless they synchronize explicitly by communication), each at its own pace, with its own notion of time. This model lent itself to an implementation by Transputer code [26] where the abstract daemons are implemented as workspace addresses. The workspace encodes, reflecting the Transputer layout for optimized use of memory, all the abstract functions which we introduced for a transparent definition of the semantics of Occam programs. We have used a similar approach for dealing with threads in a platform independent high-level definition of the dynamic semantics of Java [43]. In [78] this locality principle is pushed further to obtain a provably correct and reusable compilation scheme, from source to intermediate code, by orthogonal specification and verification of the semantics and of the translation of independent language concepts. As

[2] Later I discovered that a similar idea has been used before in numerous places in the literature, the earliest reference I found is [51] where expression evaluation is described by a processor which "moves through the ... expression tree making local transformations in it".

with attribute grammars, abstract syntax tree nodes, representing a syntactical category, are used to describe the semantics and the compilation of the involved language construct, together with the relevant context (including the data and control flow); this is enhanced by introducing generic (parameterized) ASMs to define frequently occurring and reusable transformations.

ISO PROLOG state definition. This example is taken from the ASM definition for the ISO standard of the dynamic semantics of Prolog [22, 23, 39]. The members of the ISO Prolog standardization committee wanted to see Prolog computations as depth-first left-to-right tree traversal, in search of possible solutions to the initially given query using the SLD resolution mechanism. We described therefore Prolog computation states as elements of an abstract set *NODE* coming with a static 0-ary function *root*, a 0-ary dynamic function (i.e. a function whose value will change during the computation) *currnode* for positioning the current node and a function

$$father : NODE - \{root\} \rightarrow NODE$$

which keeps track of the originator of the given step. We had to enrich this tree structure by the information each element of *NODE* has to carry for the Prolog computation state it represents, namely the sequence of goals still to be executed, the substitution computed so far, and the sequence of alternative states still to be tried. This was accomplished by the introduction of abstract domains of Prolog literals and goals (sequences of literals), of substitutions and (for a high-level description of the effect of executing the Prolog cut operator) of a set of goals "decorated" by their cutpoint information:

$$LIT,\ GOAL = LIT^*,\ SUBST,\ DECGOAL = GOAL \times NODE.$$

The goal and substitution information is associated to nodes by dynamic functions *decglseq*, *s* defined on *NODE*. For keeping track of the sequence of alternative states still to be tried, it sufficed to associate to each node an element of an abstract domain *CODE* of instruction lines containing each an occurrence of a *clause* which can be retrieved by an abstract *clause-line* function:

$$cll : NODE \rightarrow CODE, \quad clause : CODE \rightarrow CLAUSE.$$

For setting *cll* dynamically (see Figure 1), an auxiliary abstract function

$$procdef : LIT \times PROGRAM \rightarrow CODE^*$$

sufficed of which we assumed to yield the properly ordered list of code, in logic programming vernacular the candidate clause lines for the given literal in the given program. (In object-oriented terminology *procdef* could be interpreted as an array variable of the class *PROGRAM*.) Be aware that a considerable part of the standard virtual machine for implementing Prolog, Warren's Abstract

Machine, is devoted to implementing this function and the representation of terms and substitutions efficiently.

This is all one needs to know about the state (the structure defined by the abstract sets and functions) of the ASM which defines the ISO standard for the user-defined kernel of Prolog (see the details at the end of 2.1).

Abstract Instructions for Changing States. Up to now we have explained half of the freedom of abstraction offered by ASMs, namely the freedom to choose the states and thereby their signature, i.e. the number of the universes and of the operations together with their arity, domains and ranges, and the integrity constraints. The other half is the freedom to choose the machine instructions in order to reflect explicitly the actions which in the desired abstraction level are to be designed as basic dynamic actions. Through the signature we can determine which operations will be considered as given and atomic (either static or belonging to the environment), whereas the instructions allow us to determine which operations are decomposed into (multiple) basic machine steps.

What is the most general form of machine instructions to transform structures? For the sake of simplicity of exposition assume that predicates (properties and relations) are treated as characteristic (i.e. Boolean valued) functions. If the signature remains fixed, there is only one thing one can do to change such a structure[3], namely change under certain conditions the value of some functions for some arguments (usually not for all arguments and not for all functions, see the discussion of the locality principle below). Therefore the most general structure transforming machine instructions (called ASM *rules*) are guarded destructive assignments to functions at given arguments, expressable in the following form:

if *Cond* **then** *Updates*.

Cond is an arbitrary condition (statement) formulated in the given signature, *Updates* consists of finitely many *function updates*:

$$f(t_1, \ldots, t_n) := t$$

which are executed simultaneously. f is an arbitrary n–ary function, $t_1, \ldots, t_n$ are arguments at which the value of the function is set to t. We thus arrive at Gurevich's definition.

Definition of ASMs. An ASM M is a finite set of rules for guarded multiple function updates, as defined above. Applying one step of M to a state (algebra) $\mathcal{A}$ produces as next state another algebra $\mathcal{A}'$, of the same signature, obtained as follows. First evaluate in $\mathcal{A}$, using the standard interpretation of classical logic, all the guards of all the rules of M. Then compute in $\mathcal{A}$, for each of the rules of M whose guard evaluates to true, all the arguments and all the values appearing in the updates of this rule. Finally replace, simultaneously for each rule and for all the locations in question, the previous $\mathcal{A}$-function value by the newly

[3] structures without predicates, i.e. consisting only of functions, are often called algebras

computed value (if this is possible, i.e. if no two required updates contradict each other). The algebra $\mathcal{A}'$ thus obtained differs from $\mathcal{A}$ by the new values for those functions at those arguments where the values are updated by a rule of M which could fire in $\mathcal{A}$. The effect of an ASM M started in an arbitrary state $\mathcal{A}$ is to apply one step of M as long as possible (until no M-rule can fire any more because all the guards of M-rules have become false).

It may sound unbelievable, but the preceding definition is all the practitioner has to know about the semantics of ASMs in order to be able to work with them. The definition uses only basic mathematical concepts, no further theory is needed. It is adequate to use ASMs as abstract (conceptual) code; we avoid here the word pseudo-code because through Gurevich's more detailed definition [75] the abstract code offered by these machines is semantically well founded and rigorous enough to be applicable by practitioners and to be implementable, as is, by various support tools, unambiguously, without leaving room for faithful interpretations to differ between the tools or between the practitioners, whereas pseudo-code has never enjoyed this property. The at first sight astonishing simplicity of the semantics of ASMs is in part due to the fact that the only programming construct offered for free is the guarding of function updates. This explains why also application domain experts with only basic understanding of algorithmic processes can read and understand such specifications without any problem. When the system design specialist has an advantage to use more complex programming constructs (sequencing, loop, recursion, procedures, etc.), he can certainly do so, defining them in the expected way for ASMs but at the price of further semantical complications inherent in these concepts. ASMs provide not a programming, but a design language which supports linking specifications to code in a controlled (well understood and well documented) manner.

Certainly there is something more which can be said about how to use ASMs, belonging however less to their theoretical foundation than to integrating useful techniques from current system design practice. We obviously make free use of all kinds of standard programming notation (nested if-then-else, case, pattern matching, let, where, etc.) which can be reduced in a canonical way to the above basic definition. But unless there is an important reason which pays for it, we avoid the use of any more complex or non-standard concept or notation in order to keep the machines transparent and easy to follow. Usually one is only interested in states which are reachable, in the machine in question, from a given collection of initial states; the ASM approach leaves it completely open how rigorously they are defined and also imposes no particular definition language or method. Concentrating on explaining the abstraction mechanism, we have phrased the discussion in terms of one-agent or *sequential ASMs* which suffice for the purpose and by the way cover already much of practical system design (see for example how far one can go with Abrial's B-Method [3] or with the Vienna Development Method [60] which both were developed for modelling, analysing and designing exclusively sequential software systems). We discuss the extension to multi-agent (distributed) ASMs—very much needed for present-day applications—in see section 2.4., the detailed definition appears in [75].

The parallel execution model for sequential ASMs, realized by the simultaneous execution of all updates in all rules which can fire, turned out to be particularly useful in many applications. An example par excellence is provided by specifications of synchronous systems where each global clock tick is directly reflected by the application of a single machine step (which may consist in the firing of many rules); this has been used with adavantage for modelling pipelining in RISC architectures [35, 85]. The parallelism incorporated into sequential ASMs also helps to specify macro steps where one wants to hide the intended implementation by multiple lower level micro steps. It also helps to avoid premature sequentialization in system design. This is useful when the sequential execution is either not required by the logic of the problem under study or even hinders an easy to analyse high-level specification of the problem solution; in the design of a metal-processing plant [36] the abstraction, in the ground model, from a certain sequential use of the seven components of the system not only simplified the system description, but also allowed us to establish by a simple argument the strongest form of liveness which the problem formulation required to guarantee; only in the transformation of the final model to C++ code [95] became it necessary (and easy) to sequentialize the rules of the different agents. Abstraction from the sequentialization can also be crucial for achieving a modular system design when the sequential scheduling problem is difficult enough in itself to be better treated separately from the rest of the system behavior. An example is the sequential implementation of the parallelism of Occam programs in Transputer code [26]: the sequence of ASM models which links, in a proven to be correct way, Occam programs as understood by the programmer to their compilation to Transputer code, deals with the sequentialization problem separately from the Transputer details; in this case at a rather high system level where it is easy to link, in a correct and controlled way, parallel executing processes to abstract priority queues of processes managed on the basis of a time-slicing procedure.

Locality Principle. ASMs typically describe local changes (of a few functions for selected arguments appearing in the rules), the way it is needed to naturally reflect the overall behaviour of large systems as determined by local actions of some components. The computation model guarantees that in every step, everything not affected by the updates in the rules fired in this step remains unchanged. This frees the designer from having to worry about the well known frame problem; often the frame axioms are responsible for the combinatorial explosion of formal descriptions relying upon a logical or axiomatic system (or otherwise global transformation) view, as for example in Z [48], temporal logic or denotational [98] approaches. Exploiting a locality property often leads to modular design and supports stepwise refinement by offering a safe way to concentrate on the parameters which do change while neglecting the others. In contrast to a widely held view not only the system design, but also the mathematical reasoning about the system behaviour can become considerably simpler if the locality principle is used.

Consistency of updates is a problem the system designer has to care about. Obviously an execution tool should (be able to) tell, at the latest at

runtime, if and when an inconsistency does occur, as is the case in [54]. Since in general the consistency problem is undecidable, as long as we work in the high-level design area where a truly general specification method is needed, there is no other choice than to be conscious about and to try to avoid the problem by transparent design. More we move towards lower levels, more chances we have to trade expressability for decidability and for error detection by tools.

Nondeterminism can be reflected in ASMs in various ways. One can modify the basic semantical definition by firing in each step one (or some) of all firable updates or rules, which yields also a standard way to hide the consistency problem; this was the solution chosen in [74]. If one prefers to let the nondeterministic choice stand out explicitly as appearing in the signature, it can be phrased using appropriately specified selection functions. One can also express the nondeterminism using the so called qualified choose construct, introduced into ASMs by rules of the following form (with the obvious meaning) where *Rule* is an arbitrary ASM rule, D any domain of the given signature and α an arbitrary (the qualifying) condition on the choice:

Choose x **in** D **s.t.** $\alpha(x)$
Rule

Given an ASM M, i.e. a set of rules, we can express its nondeterministic variant $N(M)$ by the following ASM:

Choose *Rule* **in** M **s.t.** *firable(Rule)*
Rule

Fine tuning without formal overkill. The following real-life example is taken from an ISO standardization endeavour and illustrates fine tuning of models to the level of abstraction adequate for their intended use, meeting the desired rigor and conciseness without having to pay a tribute to formal overkill.

The example defines the kernel of the ISO standard of Prolog [22,23,39] dealing with the so called user-defined predicates P of a given program. Predicates of logic programming represent procedures which are invoqued by basic Prolog statements $P(s_1, \ldots, s_m)$, called *literals*. They are computed (in logic programming vernacular: checked to be satisfiable for the arguments $s_1, \ldots, s_n$) by trying to execute at least one of the "instructions" (called *clauses*) $c_1, \ldots, c_n$ which define the procedure code, in the order they appear in the current program *prog*. The fundamental abstraction we had to provide for the ISO definition was a mechanism to extract from the current program, for the currently examined literal (the activator *act*), this properly ordered list of candidate clauses without mentioning any detail for the implementation of this mechanism (and of the related representation of terms, substitutions, literals and clauses in the WAM). The problem is not as innocuous as it appears because Prolog programs are dynamic, modifiable at runtime. Nevertheless it is easily solved by the dynamic function *procdef* introduced above whose behavior is determined referring only to the current activator *act* and the current program (and indirectly also to the dynamic abstract association *clause* of *clauses* to lines of *code*).

Using *procdef*, the entire Prolog kernel for user-defined predicates is easily defined by adaptating, to the Prolog environment, the depth-first left-to-right tree creation and traversal, see Figure 1. The *Success* rule terminates the computation when the current goal sequence has been emptied. The *Continuation* rule makes the computation *proceed*, upon successful computation of the current procedure body (read: satisfaction of all the literals of the leading *goal* attached to the current node), by updating the goal sequence. The *Call* rule calls the procedure for the currently computed literal *act* by attaching to the current node the sequence of candidate nodes for the possible procedure body subcomputation trees, one tree root t_i for each alternative clause c_i found for the activator in the current program, and passes control to the *Select* rule for trying these children in the indicated order for execution (read: logic programming resolution).

Success

if $decglseq(currnode) = []$
then $stop := Success$

Continuation

if $goal(currnode) = []$
then *proceed*

Call

if $is_user_defined(act)$
  & $mode = Call$
then extend *NODE* **by**
$t_1, \ldots, t_n$ **with**
$father(t_i) := currnode$
$cll(t_i) := c_i$
$cands(currnode) := [t_1, \ldots, t_n]$
$mode := Select$
where $[c_1, \ldots, c_n]$
  $= procdef(act, prog)$

Select

if $is_user_defined(act)$
  & $mode = Select$
thenif $cands(currnode) = [\,]$
then *backtrack*
else *resolve*
where $backtrack \equiv$
  if $father(currnode) = root$
  then $stop := Failure$
  else $currnode := father(currnode)$
    $mode := Select$

Fig. 1. ISO Prolog kernel (for user-defined predicates)

The function *goal* yielding the leading goal of a Prolog state is a derived function, i.e. a function which can be defined in terms of other functions, here explicitly in terms of *decglseq* and of list functions. Similarly for *act* where the definition incorporates the scheduling optimization for determining which one of the literals in the leading goal is selected for the next execution step. Even at the level of abstraction we are considering here, namely of the semantics of the language, the definition of the abstract action *resolve* is a matter of further refinement. It provides the really logical ingredients of Prolog computation steps, involving the logical structure of terms and the logical concepts of substitution and unification. Similarly the definition of *proceed* provides further insight

into the tree pruning and the error handling mechanisms which enter ISO Prolog through the extra-logical built-in predicates *cut* and *throw* (see[22, 39] for details).

The four rules of Figure 1 define the abstractions which turned out to be appropriate to clarify the database update problem which has intrigued for a long time the ISO standardization effort [25, 38].

2.2 Building Ground Models

In the introduction to [3] Jean-Raymond Abrial states "that the precise mathematical definition of what a program does must be present at the *origin* of its construction" with the understanding that the concept of program incorporates that of specification. For epistemological reasons the most difficult and by statistical evidence[4] the most error prone part of the thus understood program construction is at the origin because it is there that we have to relate the part of the real-world under study to the models we have for it in our language (representing the models in our mind). Since there is no infinite reduction chain between models, as discussed already by Aristotle[5] criticising Plato [8], this link itself cannot be justified theoretically, by mere logico-mathematical means of definition or proof. But still, in order to gain confidence in the models which are the basis for the entire program construction leading to executable code, we must justify their appropriateness by connecting somehow their basic objects and operations to the basic entities and actions we observe in the real world.

Pragmatic foundation is the best we can achieve, exploiting conceptual and experimental features of the models. A conceptual justification can be given by grasping the adequacy of the *ground* model [6] with respect to the informal description of the world, through direct comparison. This requires that the ground model is tailored at the level of abstraction at which we conceive the part of the real world to be modelled, i.e. in such a way that its mathematical objects, predicates, functions, transformations correspond in a simple way, possibly one-to-one, to the entities, properties, relations, processes appearing in the informal "system requirements" to be captured. An experimental justification can be provided for the ground model by accompanying it with clearly stated system test and verification conditions (system acceptance plan), i.e. falsifiability criteria in the Popperian sense [109] which lay the ground for objectively analyzable and repeatable experiments. This too requires an easily inspectable, comprehensible and transparent link between the formal ground model and the informal problem description, the outcome of the experiments having to be confronted

[4] 80% of the errors in system programming do occur at the level of requirement specifications.

[5] "Every theory and in general every deductive knowledge has a certain wisdom as premise."[6]

[6] In [20, 39] they were called *primary models* to stress that they are not in any way absolute or unique but simply starting points for a series of mathematical transformations.

with the informally described real-world situation (see below the discussion of the oracle problem of testing and more generally of validating high-level models by simulation). In comparison to the conceptual and experimental justification of a ground model, the mathematical justification of its internal consistency is the smaller although not negligible problem, essentially a problem of high-level reasoning and (where possible machine assisted) proof checking.

Finding the right abstraction level is the main problem the definition of appropriate ground models has to face. One must discern the aspects of the desired system which have to be included in the mathematical model, for its being semantically complete, satisfying both the customer and the designer, and relegate the ones not relevant for the logic of the problem to further design decisions, as belonging to the implementation. This choice can be made and justified appropriately only on the basis of the related application domain knowledge, typically in close cooperation between the system designer and the application domain expert who brings in the informal requirements. Usually these initial requirements are neither complete nor minimal and it needs engineering and conceptual skill to distill the relevant features, finding those which are lacking in the informal description (because implicitly assumed by the domain expert) and hiding the dispensable ones, reserving them for further refinement steps. The need for cooperation between the user and the designer to construct the ground model reveals yet another difficulty, namely to have a common language for sufficiently unambiguous communication between the two parties, as is confirmed by statistical data: two thirds of the software development time are spent for communication between user and designer; one quarter of software project failures is due to user/designer communication problems; mismatch in system requirements understanding between user and designer is recognized as the most frequent cause of user dissatisfaction with the final product.

ASMs solve the language problem, for communication between customer and designer, by using only common mathematical and algorithmic concepts and notation for the description of real world phenomena. They also contribute to satisfactory and practically viable solutions of the formalization problem. The freedom of abstraction allows the system designer to model the informal requirements by expressing them directly[7], in application domain terms, without having to think about formal encoding matters which are extraneous to the problem. In the VDM approach, before starting the modelling work, one first has to learn "the most basic kinds of data value availabe to the modeller and ... how values can be manipulated through operators and functions" [60, p.72] and is then forced to formalize everything in terms of these fixed data abstraction possibilities, inventing encodings if they don't fit directly. With ASMs the designer can freely choose the basic data values (objects of abstract domains) and their manipulations (through functions or updates), looking only at what is showing up in the application domain and in the problem explanation given by the cus-

[7] "A significant engineering project begins with a specification describing as directly as possible the observable properties and behaviour of the desired product".[81, p. 4]

tomer. Similarly the freedom to separate modelling from justification concerns and, when it comes to proving, to choose the appropriate level of proof, solves also the justification problem. Mathematical experience as well as experience with machine assisted proof systems show that for the purpose of proving, one often has to strengthen the claim and to provide more details than what appears in the property to be proved. Specification *tout court*, or coming with a high-level instead of a formalized and machine checked reasoning, also represents a form of abstraction. True, "when the proof is cumbersome, there are serious chances that the program will be too" [3, p.XI]; but equally well too many details imposed by the proof system may unnecessarily complicate the design.

Thus an ASM ground model has the chance to serve equally well the customer and the designer. It can be understandable for the customer who typically is not thinking in terms of data representation and usually is not trained to understand proofs. The designer can show to his team of software experts that it is consistent and satisfies the required high-level system properties. In addition it can be sufficiently rigorous and complete for the design team to proceed with mathematical analysis and transformation into an implementation.

Supporting practical design activity. ASMs enable the designer to make his intuitive ideas explicit by turning them into a rigorous definition which, far from being an "add-on", documents the result of his "normal" activity and thereby provides the possibility of an objective assessment of the properties of his design. The abstraction freedom does not provide the intuitions needed for building a good ground model, but it permits to smoothly accompany their natural formulation, turning them into a concise definition, with an amount of writing and notation proportional to the complexity of the task to be specified; differently from most other formal methods, ASMs enable us not to complicate matters by formal overhead, simply because no (linguistic or computational) modelling restriction forces us to include (encoding or scheduling) details which belong only to the specification framework and not to the problem to be solved.

As with every freedom, the abstraction freedom puts on the designer the full responsibility for justifying, to his team of programmers and to the application domain expert, the choices made for the basic data (domains with predicates and functions) and actions (rules). This leads him naturally to state and collect along the way all the assumptions made either by the customer, for the system to work as desired, or by the designer, for laying out the software architecture in the ground model. This helps not to depend, at least not inadvertedly, upon any bias to particular implementation schemes and has the important practical effect to make relevant application domain knowledge explicitly available in the ground model documentation.

These features of ground model development can be illustrated by two examples in the literature [36, 17] where an informal requirement specification has been turned into a ground model and implemented, going through stepwise refined models, by C++ code which has been validated through extensive experimentation with the provided simulators. The robot control example [36] exploits the ground model abstractions for guaranteeing, in a straightforward manner,

all the required safety properties, under explicitly stated high-level assumptions which could easily be established to be true for the subsequent refinements; similarly the ground model abstractions allowed us to establish the strongest form of the required liveness (that "every blank inserted into the system will eventually have been forged") and the maximal performance property for the system. All these properties could be verified by a standard analysis of the ASM runs, carried out in terms of traditional mathematical arguments and later also mechanically confirmed by a model checking translation in [132]. In both examples the ground model refinements have been developed up to a point from where the executable C++ code could be obtained through an almost mechanical translation of ASM rules into C++ procedures which are executed in a context of basic routines implementing the ASM semantics. This illustrates the role of intermediate models for documenting the structure of the executable code as well as the design decisions which have led to it; these formally inspectable intermediate models can serve as starting point for possible modifications coming up in the code maintenance process (optimizations, extensions, etc.).

The freedom of linguistic and computational abstraction makes ASMs also adaptable to different application areas and yields easily modifiable and in particular extendable models. See for example the ease and naturalness with which the models for Prolog and its implementation on the Warren Abstract Machine [39,40] could be extended post festam for including polymorphic types or constraints and their implementation on corresponding WAM extensions [15,16,41, 42], see also the WAM extension by parallel execution with distributed memory in [5] and the adaptation to the implementation of scoping of procedure definitions in a sublanguage of Lambda-Prolog where implications are allowed in the goals [90]. Some students in my specification methods course in Pisa in the Fall of 1998 have produced an almost straightforward after the fact extension of the production cell ASM in [36] to the fault-tolerant and to the flexible real time production cells proposed in [93,94], another group has adapted the high-level steam-boiler ASM [17] for a modelling of the emergency closure system for the storm surge barrier in the Eastern Scheldt in the Netherlands [97].

2.3 Refinement Technique

It is nowadays a common place that software design has to be hierarchical and has to be based on techniques for crossing the abstraction levels encountered on the long way from the understanding of the problem to the validation of its final solution. There are numerous proposals for defining and relating these levels as support for the separation of different software development concerns: separating program design from its implementation, from its verification, from its domain dependence[11], from hardware/software-partitioning [91], system design from software design from coding and similarly for testing (Sommerville's V-model [121]), functionality from communication, etc. Numerous vertical structuring principles have been defined in terms of abstraction and refinement.

Semantical refinement is what ASMs add to this, due to their freedom of abstraction, refinement being the reverse of abstraction. Therefore the designer

can follow the needs for vertical structuring and separation of concerns (including modularization and information hiding) as suggested by the semantics of the system to be developed, without being directed by any a priori fixed syntactical or computational boundary condition one would have to comply with. He is also not forced to first learn one of the many complicated but typically rather specialized and not easily applicable refinement theories in the literature. Since the abstraction mechanism incorporated into the definition of ASMs is as general as the present day scientific knowledge enables us to conceive (see section 3.), the corresponding ASM refinement method permits to push Dijkstra's [57] and Wirth's [134] refinement program to its most general consequences. ASMs allow us to realize it independently from the restrictions which necessarily come with every concrete programming language for executable code, producing maximal practical effect for hierarchical and ideally provably correct system design.

Various practical refinement notions have been developed by us, through real-life case studies, in an attempt to close, in a controllable manner, the gap between the design levels involved. They can all be put into the form of the well known commuting diagram of Figure 2, for a given machine A which is refined to a machine B, where a usually partial *abstraction function*[8] $\mathcal{F}$ serves as proof map, mapping certain refined states $\mathcal{B}$ of B to abstract states $\mathcal{F}(\mathcal{B})$ of A, and certain sequences R of B-rules to sequences $\mathcal{F}(R)$ of abstract A-rules (in cases where the proof map is used as *refinement function*, $\mathcal{F}$ goes from A to B).

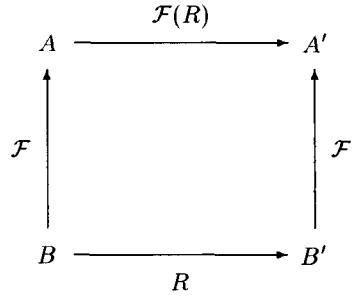


Fig. 2. ASM refinement scheme

In order to establish the desired *equivalence* of the two machines, before proving the commutativity of the diagram, one can (and first of all has to) *define* the appropriate notions of *correctness* and/or *completeness* between refined runs $(\mathcal{B}, \mathcal{S})$ and abstract runs $(\mathcal{A}, \mathcal{R})$. This definition is in terms of the locations (the "observables") one wants to compare in the related states of the two machines. The observables could be, for example, the operations the user sees in the ab-

[8] Schellhorn [115] generalizes this to relations and provides examples where it is convenient to relax the abstraction and refinement functions to relations.

stract machine, which are implemented through the refinement step. This case is characteristic for the refinement concept of the B-method: "During each such step, the initial abstract machine is entirely reconstructed. It keeps, however, the same operations, as viewed by its users, although the corresponding pseudo-code is certainly modified" [3, p.XVI]. ASMs offer an a priori not restricted spectrum for instantiating the refinement notion, for fine tuning it each time to the underlying implementation idea; this allows one to concentrate on the main problem, namely to find and to describe in an understandable way the appropriate design idea, i.e. "the right $\mathcal{F}$", the function (or relation) which incorporates and documents the software engineering experience and implementation skill. The difference of the refinement concepts for B and ASMs is probably a consequence of the fact that in B, "in the intermediate refinement steps, we have a hybrid construct, which is not a mathematical model any more, but certainly not yet a programming module" [3, p.XVI], whereas the refined ASMs are certainly mathematical models, the same way as the abstract ones.

We call $\mathcal{F}$ a proof map because it guides the justification the designer provides for the semantical correctness of the refinement step. Proofwise the ASM refinement notions are supported, in their full generality, in PVS [58] and KIV [115]; Schellhorn defines sufficient conditions for correctness and/or completeness preserving composition of commutative refinement diagrams out of subdiagrams, which yield a modular scheme for provably correct refinements of finite or infinite runs. This confirms our experience that it helps if one provides many and easily controllable refinement steps instead of only a few but difficult ones; as a by-product of such refinement sequences one obtains a detailed documentation for each single design decision, a feature which supports design-for-reuse, easy extendability and reliable maintenance of the final product.

Numerous case studies illustrate the far reaching practical use of semantically oriented refinement chains, in so different domains as implementation of programming languages, architecture design, protocol verification and development of control software. The 12 intermediate models created in [40] to link in an easily justifiable way the ISO Prolog model to its implemementation on the WAM served to implement the following features: the backtracking structure (in a stack model with reuse of choicepoints), the predicate structure (introducing determinacy detection—a look-ahead optimization—and compiling the selection of alternatives by (re)try/trust code with switching), the clause structure (implementing continuations by environment (de)allocation and the clause compilation by unify/call code), the term and substitution structure (introducing the heap, implementing the unification, compiling clause heads/bodies by getting/putting instructions and variable binding) and WAM optimizations (environment trimming, last call optimization, local/unsafe variables, on-the-fly initialization). There is not a single step which has been suggested or guided by the syntax of Prolog programs. It turned out later that these intermediate models make it possible to extend without difficulty both the specification and the proof steps, from Prolog to Prolog with polymorphic types or constraints

and their implementation on the corresponding WAM extensions PAM [15,16] and CLAM [41].

The refinement hierarchy in [27,26] led to a provably correct compilation scheme for Occam programs to Transputer code. It reflects standard compilation techniques, some peculiarities of the communication and parallelism concept of the language and some characteristic Transputer features. There was only a small contribution from program syntax to the following chain of 15 models for the implementation of the following features: channels, sequentialization of parallel processes (by two priority queues with time-slicing), program control structure, environment (mapped to memory blocks addressed by daemons), transputer datapath and workspace (stack and registers, workspace and environment size), relocatable code (relative instruction addressing and resolving labels), everything with a high-level notion of expression and expression evaluation which abstracts from the peculiarities of the Transputer expression evaluation.

The chain of over 15 models leading from Java through its compilation on the JVM to the full JVM [43,44,46] reflects two goals. One is related to the language and its implementation on the JVM, namely to orthogonalize sequential imperative (while program) features, static class features (procedural abstraction and module variables represented by class methods/initializers and class fields), object oriented features, the error handling mechanism and concurrency (threads). The other goal is related to the Java Virtual Machine as a Java independent platform, namely to separate trustful bytecode execution, bytecode verification and dynamic loading concerns in order to make their interaction transparent.

The equivalence notions defined for the commutative diagrams in such compilation scheme correctness proofs prove much more than the traditional equation

$$[P]_{source} = [compile(P)]_{target}$$

where the square brackets denote the denotational input/output program meaning. Typically this meaning is defined by a fixpoint solution to certain equations over abstract domains, following the syntactical structure of P. A bitter consequence is that applications of such domain based methods are usually restricted to relatively simple properties for small classes of programs; the literature is full of examples: *pure* versions of various functional programs (pure instead of common LISP programs [108]), Horn clauses or slight extensions thereof [123] instead of Prolog programs, structured WHILE programs [100] instead of imperative programs appearing in practice (for example Java programs with not at all harmful, restricted forms of go to). A remarkable recent exception is the use of denotational semantics for the functional-imperative language ComLisp in the Verifix project [131]. The add on of ASMs with respect to denotational methods is that properties and proofs can be phrased in terms of abstract runs, thus providing a mathematical framework for analyzing also runtime properties, e.g. the initialization of programs, optimizations, communication and concurrency (in particular scheduling) aspects, conditions which are imposed by implementation needs (for example concerning resource bounds), exception handling, etc.

The refinement step in [32] is completely unrelated to program structure. It is determined by the intention to separate the considerations one has to make for durative actions from the correctness concern for the mutual exlusion protocol with atomic actions. This algorithm design in two steps simplified considerably the analysis of Lamport's protocol in the literature. The five design levels introduced in [35] serve to separate and justify the major techniques involved in pipelining a typical RISC microprocessor, namely parallelization and elimination of structural, data and control hazards. The refinement steps in [36, 17] are guided by the intent to turn the given informal problem description into a well documented and justified to be correct solution of the problem by C++ code; each of the intermediate models reflects some important design decision.

2.4 Decomposition Technique and Function Classification

(De)Composition provides horizontal structuring of systems as a tissue of separate subsystems which interact with each other through well defined interfaces. For decomposing a system one has to recognize, within the global system behavior, separate roles and to encapsulate them into the definition of subsystems. Thereby the strength of a decomposition method is related to its abstraction capabilities for defining each component's behavior and its interface for the interaction with other components, with the necessary transparency and precision, conciseness and completeness, abstracting from internal details. The freedom of abstraction the designer enjoys with ASMs offers the corresponding freedom to device rigorous system (de)composition techniques which are not limited to notational (signature) issues, as large parts of UML [130] are, but can handle problem oriented modularization by reflecting abstractly semantical component content (functionality) and interface behavior (interaction content). We explain in this section that ASMs not only support best modularization practice, but also enhance it by lifting it from programming to rigorous high-level design and extend it to the case of multi-agent systems.

Best modularization practice needs a most flexible abstraction mechanism[9]. The **ASM function classification** realizes a rigorous high-level concept of modularity which leaves more design freedom than (but can be specialized to) the modularization and compositionality principles in current programming languages. In an ASM M we distinguish *basic* functions from *derived* functions (which are defined in terms of basic ones). Within derived or basic functions we separate *static* functions (which remain constant during M-computations) from *dynamic* ones; among the dynamic functions we distinguish—using a terminology appearing in Parnas' Four Variable Model [107]—the *controlled* ones, which are subject to change by an update appearing in a rule of M, from the *monitored* ones which can change only due to the environment or more generally due to actions of other agents. Last but not least there are *shared* functions which can

[9] "A module achieves program simplification by providing an abstraction. That is, its function can be understood through its interface definition without any need to understand the internal details."[96]

be written by M and by some other agent and for whose consistency typically a protocol has to be devised. Shared functions help to naturally reflect multi-agent computations and combined read/write use of locations (like ports in chip design which are used for both input and output).

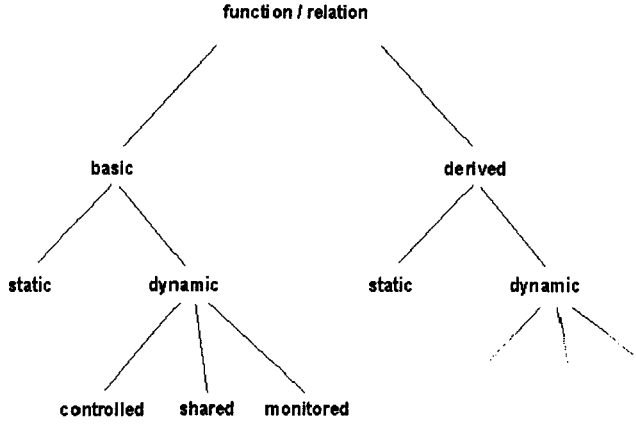


Fig. 3. ASM Function Classification

Distinguishing between basic and derived, static and dynamic or controlled and monitored functions constitutes a rigorous high-level realization of Parnas' **information hiding** principle achieving "designer control of the distribution of information" [103, p.344] through the extent to which the description of abstract non-controlled functions is given to the programmer. A function may be described by its signature only (conveying only syntactic or type information), by an implicit axiomatic definition (through possibly semantical constraints describing "the what without the how"), by an explicit or recursive definition (describing the mathematical law which determines the semantical meaning of the function modulo the occurring basic functions), by a module interface definition, by an algorithm (fixing also the procedure by which the function values are computed), by another ASM or by a detailed program (giving away all the implementation details). The programmer has full control (read and write access) only upon the controlled functions, the ones which he is asked to program, but for accomplishing this task he is freed from the obligation to also care about how to define the non-controlled functions which he can read without any restriction when determining the arguments and the new values for updates of controlled functions, and upon whose effect he can rely as much as he has been informed.

Non controlled functions allow the designer to specify and reason about his system on the basis of what is given through such functions. This is not a phenomenon of underspecification of the system nor does the system become fuzzy

through the possible lack of detailed information on the non controlled functions. Whenever, in a run of an ASM M, a non controlled function e is invoked in a state S, it appears in S as a given function and thus can be freely used, although we may not know (or may want to abstract from) how the current interpretation of e, in S, has become available to us. For example in an update $a := f(a, g(b))$ (with say g declared as a unary monitored function and b as a static 0-ary function), the term $f(a, g(b))$ is used to denote an object in state S, obtainable through a computation (the standard logical interpretation) which follows the term structure and uses the interpretation of a, b, g in S which is *known* once the state is given, never mind how these interpretations have been defined. It is indeed only conceptually that we need to (de)compose our systems, therefore for determining the value of the complex construct $f(a, g(b))$ in S we do abstract from the way the monitored function g and the static function b have been given in S but nevertheless have the functions available for use.

Numerous **examples** illustrate the power of information hiding and modularization through non-controlled functions in ASMs. In the model for the IEEE VHDL'93 standard [30,31] the details of the propagation of signal values in zero time are hidden from the definition of the VHDL kernel by two derived functions for the so–called driving and effective values (which we define by a recursion on the signal sources resp. on port association elements from ports to signals, abstracted from the complex algorithms in the language reference manual [86]); similarly the availability of values in zero time, at certain points in a circuit, can be described by a static (or derived) function, usually expressing a combinatorial network. The details of error propagation and handling in the Java Virtual Machine are separated in [44] from the main machine definition by a derived, recursively definable *catcher* function. The details of the unification procedure have been encapsulated in [39] into a static function *unify*. The reactive behaviour of (concurrently operating) PVM daemon processes, in response to requests coming from local tasks to carry out some PVM instruction or to the reception of a message from another daemon, is modelled in [28,29] using a monitored function *event* whose values *event(daemon)* are assumed to remain stable until *daemon* has read the function destructively; this abstracts from the details of a communication scheme. In [43] compilation features are separated from the dynamic semantical aspects by encapsulating them into static functions; the implementation defined scheduling algorithm, which resides on top of the synchronization of Java threads as required by the language reference manual, is encapsulated into a monitored function. Derived and monitored functions can also been used for modelling the interface between the discrete and the continuous world in hybrid systems. An example is the real-valued speed function appearing in the high-level Falko-ASM, developed at Siemens within a project for building a tool for train schedule development and validation. Train speed has been encapsulated there into a derived function which is computed from monitored data, using the laws of physics and continuous mathematics.

Also syntactical composition principles can be put to fruitful use in ASMs. An example is the composition of a VLSI implemented microprocessor out of

basic architectural components (sub-ASMs) in [24]; it provided the possibility to analyse processor properties by reducing them to properties of the components and has been the key for accomplishing a reverse engineering project for the microprocessor control of the APE100 massively parallel machine. In [55] this approach is exploited further for supporting hardware/software-partitioning and instrumenting of building blocks. Another example is the submachine concept, illustrated by the replacement in [40] of the above mentioned function *unify* by a sub-ASM which implements a unification algorithm. A promising scheme for composition of ASMs out of generic (parameterized) ASMs, supporting library building and reuse of components, has recently been defined analysing transformations which occur typically in programming language compilation [78].

The flexibility ASMs offer for rigorous high-level definitions of system components and their interaction facilitates a transparent component reuse and helps to master the compatibility problem for the design of heterogeneous systems; a successful industrial use of this property is reported in this volume [89]. The abstraction possibilities also help keeping the global understanding of a system during its entire development, from the ground model and through the refinement definitions down to the implementation. Statistical evidence and practical experience reported in [10] confirm the importance, for a software project to succeed, of maintaining such a global system view and of the possibility to make it accessible through an appropriate documentation. We made the same experience when using ASMs for documenting the basic functionality of a large C++ software package, developed and used successfully at Siemens for the simulation of railway systems. The problem consisted in specifying the components of the system at a level of abstraction appropriate to make the constraints about the component interaction explicit [102].

Distributed (multi-agent) ASMs, where multiple sequential ASMs operate concurrently, support the decomposition in a particularly important way. Gurevich [75] has defined a notion of multi-agent ASM run which avoids any commitment to particular forms of concurrency, distilling the bare minimum needed for guaranteeing the global consistency of local states. It says that such a run is a partially ordered set M of "moves" x of a finite number of sequential ASMs (agents) $A(x)$ in which

1. each move has only finitely many predecessors,
2. the moves of every agent are linearly ordered,
3. each initial segment X corresponds to a state $\sigma(X)$—the result of executing all moves in X—which for every maximal element $x \in X$ is obtainable by applying move x in state $\sigma(X - \{x\})$.

This definition guarantees what is needed in applications, namely that the scheduling of moves which are independent of each other in the given run does not influence the (global view of the) state resulting from the run, more formally expressed: given any finite initial segment of a run, each linearization has the same final (global view of the) state. This provides the freedom needed to faithfully reflect distributed systems as they occur in practical life without be-

ing committed a priori to special synchronization concepts, in particular not to framework dependent timing conditions.

The most important (large) example one can point to is the forthcoming distributed real-time model for the to be defined SDL-2000 standard [64,66] which provides a rigorous but concise and extensible definition for the practitioner's intuitive understanding of the functional and timing behavior of SDL. In addition to the model of Basic SDL-92 in [65] (where the behavior of channels, processes and timers with respect to signal transfer operations is formalized following faithfully the International Telecommunication Union *T Recommendation Z.100*), it provides structural system decomposition in terms of concurrent processes (ASM agents) which interact communicating asynchronously through gates for signal exchange (their interfaces). Another (small) example is the above mentioned robot controller [36] where the concurrently operating components (sequential ASMs) are small finite automata with some additional state structure and interface conditions. We use there shared functions as interfaces which, where necessary, make the sequentialization conditions for certain actions (of the otherwise independent components) explicit and at the same time describe abstractly the effect of these actions. These interfaces simplified considerably the specification and the amount of work necessary to establish the required safety and liveness conditions. Our interfaces can be naturally mapped to the CSP-style synchronization mechanism in [114], a particular communication scheme which has been encoded in Concurrent ML.

2.5 Analysis and Integration Techniques

In this section we survey the practicability of the method which is due to the simplicity of ASMs (making them easy to read and to write for the practitioner) and to the support and enhancement they provide for existing system design and analysis techniques.

Naturaleness. ASM system design comes as a natural thing for the practitioner. Differently from what happens with most academic approaches, the system programmer is not asked to be converted from the common process and state oriented model based reasoning. In the contrary the operational view is enhanced by adding fine tuning, to whatever abstraction level is appropriate for the application under hand, and by putting all this on a simple and rigorous foundation. The reader has seen above the definition—which starts from scratch and is all one needs to understand for applying the method. It is what I had (see [73]) when, as a logician without any practical experience, I joined Gurevich to explore his bold ASM thesis [76]. A frequent critique of formal methods complains about the need for extensive specific training in these methods before they can be put to fruitful industrial use, if they are applicable at all: "formal methods cannot be ... a distinct "add-on" that goes beyond the "normal" things that software developers already do" [105, p.195]. The ASM approach supports directly, without need for special training, the software developers' daily work and improves its quality by enabling the engineer to "produce documents that

are easier to read than the programs that they describe" [105, p.195]. Admittedly there is one difficulty, something one has to learn, namely to find the right abstractions. But this is a problem *in re*, of system design, not of the adopted method. ASMs are different from most other methods in that to this unavoidable difficulty they do not add any formalistic complications concerning the design language, the model of computation, the analysis and reasoning schemes, etc. The combination of conceptual programming and rigorous reasoning, characteristic for ASM modelling, directly supports engineering skill and experience, leading from ground models via refinements to implementations.

Integratability. The ASM approach is not monolithic but provides a well defined semantical basis for integrating other description techniques or synthesis methods (see section 3.2) and can itself be integrated, at any development level, into established design flows. The integration potential is technically achieved through committing to standard mathematical and programming language, concepts and notation and through the use of abstract functions; more specifically through the freedom to choose how and to which degree of precision derived, static and monitored functions and integrity constraints are determined (axiomatically, algebraically, functionally, algorithmically, etc.). Due to such functions ASMs represent "code with holes", the important point being that the looseness (desired implementation freedom) of the model is circumscribed by the description of the holes.

This versatility also facilitates tailoring the general ASM specification mechanism to typical data structures and computational patterns which appear again and again in a given application domain. The semantical simplicity and openness of ASMs result in their flexibility for incorporating special application domain knowledge, namely by adapting the general method to the particular structures of that domain. The specialization helps to elaborate the application domain knowledge in a tool based way, thus making the ASM specifications and transformations reusable. An excellent illustration is provided by the Verifix project [131] where the use of ASMs and their refinements—for the semantical definition of source, intermediate and target languages [136]—is coupled with numerous well established or advanced compiler generation and verification techniques (see for an example [68] in this volume) in order to achieve the overall project goal, namely to build a correct compiler for a real-life high-level language on a real-life machine (DEC-Alpha processor).

Verification Techniques. As a consequence of this conceptual openness and of separating design from verification concerns, the method combines problem oriented modelling and simple design with the applicability of state-of-the-art analysis techniques for validation and verification (by "head" or by machine, interactively or fully automated). Most importantly for the practitioner's daily work, ASMs support on the fly justification by the designer because the intuitions and reasoning about the system behavior, which indeed guide and accompany the design, can be turned into rigorous statements about runs and can therefore be checked objectively (refuting or justifying them exploiting traditional mathematical methods). In this way ASMs not only help to debug the designer's work

from its very beginning, but have also the important practical effect of producing an intersubjective documentation which makes available to the rest of the world what the designer had in mind when building the system. The key for the success of this program, which leads much further than purely static analysis methods, is that ASM models allow us to abstract, to rigorously formulate and then to verify or validate runtime properties through analysis or simulation.

In certain (theoretical and industrial) circles it has become fashionable to claim that mathematical proofs are not reliable and have to be replaced by machine checked proofs. Without reentering this discussion here (see [21]), it should be pointed out that nothing prevents from verifying standard mathematical reasoning about ASMs by interactive or fully automated proof tools which force us to make explicit, for the checking machine, all the details which good mathematical proofs are geared to hide in order to convey to the human reader a transparent picture of the proof. The additional effort needed for proof checking ASM properties by machines is similar to the additional effort needed to implement ASMs: the abstract domains have to be represented by standard data structures of the prover, the static functions have to be defined completely by equations, axioms, etc.. Delivering our work to the prover forces us to cash the freedom of the high-level reasoning; for this endeavor it is helpful to find and exploit similarities between the proof structures and the design structures. During the last years various such investigations have been started and some encouraging results are already reported in the literature. The correctness proof in [40], for the compilation scheme from Prolog to WAM code, has been verified in KIV [116] (and for some of the proof steps also in Isabelle [111]). The correctness proof for the first refinement step in the model for pipelining a RISC machine in [35] has been checked in KIV [67] and in PVS [124]; the proof checker applied in [79] has discovered an omission of a hazard case in the last refinement step. The properties proved for the production cell ASM [36] have been successfully model checked in [132]; this work is part of an effort to exploit the abstraction possibilities of ASMs to contain the state explosion problem when designing FSMs for model checking. The ASM proofs used in [136]—showing the correctness of bottom-up rewriting specifications for back-end compilers from intermediate languages into binary RISC processor code—have been checked using PVS [58].

Validation Techniques. The method enhances current industrial system design through the possibilities for early and high-level validation of ASMs, whether of system components, of their interaction through interfaces, of particular requirements in the ground model, etc.. We do not want to enter the discussion in the literature [77,61] whether specifications should be executable or not. There are basically two complementary possibilities to simulate high-level ASMs. One is through a general purpose simulator which can execute any given ASM once its non-controlled functions are defined (interactively, by inputting scenarios, by ad hoc implementations, by using library functions, etc.). An example is the workbench presented in this volume [54] which has been used successfully for extensive simulation of various versions of the above mentioned high-level Falko-ASM. An ASM M can also be made executable by a special pur-

pose interpretation or compilation which exploits the programming environment of the application domain of M for implementing the non-controlled functions. The Prolog kernel ASM in Fig.1 has been implemented at Quintus and used there for various experiments with Quintus Prolog [47]; it took a couple of hours to write this simulator where the two non-trivial non-controlled functions, *unify* and *procdef*, have been linked to available Quintus code. During our work on the model for the JVM in [44], Wolfram Schulte has realized and tested successfully an executable version of it by implementing in Haskell the recursive definitions for the static functions. On the basis of a recent KIV implementation of the Java-ASM in [43], all the Java programs appearing in [69] have been tested successfully [125], except those which lead to compilation errors (due to type check problems, etc.) or deal with arrays, two features we did not cover in our Java model. Another group of examples uses a direct encoding of (refined) ASMs into C++, done ad hoc [36] or using a compilation method (see [17] and Joachim Schmid's recent transformation of the Falko-ASM into C++ code). In between these two approaches we find the special-purpose simulator Gem-Mex developed by Matthias Anlauff to make ASM interpreters for a class of programming languages executable [4]. The tool exploits the ASM specification techniques for the typical structures occurring in the definition and compilation of imperative programming language constructs. This idea has been used and further developed in the Verifix project [136, 78] and represents an instructive example for combining the advantages of the general ASM approach with those coming from the specifics of application domain structures.

3 Universality of ASMs for System Development

In [76] Gurevich gives a penetrating epistemological justification of his strengthening of Turing's thesis which stood at the origin of the discovery of the ASM concept [71, 72], namely that every sequential computational device, on any level of abstraction, can be simulated in lock-step by a sequential ASM of appropriately the same size. We collect here some empirical evidence that this thesis is the practical system design analogue of Turing's thesis.

The ASM thesis is confirmed by the following four experiences. First to be mentioned is the multitude and diversity of successful complex system design and analysis projects, carried out using ASMs and covering the definition of the semantics of real-life programming languages and their implementation, real-time algorithms, protocols, control software, virtual machines and architectures, see [33, 2]. Second to be mentioned is the adaptability of ASMs to whatever application domain for solving the ground model problem. Thirdly the ASM method completes, for practical system design, the ambitious structured programming project, synthesizing in a remarkably simple way two fundamental lines of thought in the history of ideas in computer science; see the historical details in section 3.1. Last but not least other well known design and computation models are naturally embedded into ASMs where they can be recognized by specializing the signature, the rules, the constraints, the runs. We show this

below for VDM [60], Abrial's Abstract Machine Notation [3], Parnas tables [107], stream X-machines (an important subclass of Eilenberg's X-machines [59]), Petri nets [113] and Codesign FSMs [91]. ASMs appear to be a truly encompassing framework for successfully comparing and combining major current design and analysis techniques, on a rigorous semantical basis which allows one to exploit their respective advantages by appropriate specializations and restrictions.

3.1 Abstract Machines + Abstract State = ASM

The term *abstract machine* has been coined in 1968 by Dijkstra [56], in the context of defining the operating system T.H.E., preceded by the class concept of Simula67 which has been interpreted as a form of abstract machine [80, 52]). Numerous variations appear in the literature, like hierarchical systems, layered architectures, data spaces, virtual machines [106,137,84,126,51], with corresponding program development methods like top-down design, multi-level programming, stepwise program refinement and data abstraction, etc., which characterize the structured programming and abstract data type method [134, 53] and have prepared the ground for VDM [60] and for Abrial's sophisticated combination of Abstract Machine Notation with proof controlled stepwise refinements [3].

Nevertheless nobody has addressed the epistemological and practically important question of how far the proposed concept of abstract machine can lead us or whether there is at all a rigorous most general notion of abstraction determining the underlying states and operations of such machines. This is even more surprising given that the development of the data abstraction idea in the theory of abstract data types has led to understand that a most general notion of state does exist [101,62], namely Tarski's logical concept of structure [127]. However in the theory of abstract data types, structures have been encoded into terms so that equational or axiomatic algebraic specification methods could be applied—as a bitter consequence state changes, instead of being related to the dynamics of machine instruction execution, remained static, solutions of fixpoint equations. This approach has its mathematical beauty but turned out to be impractical.

Although the two ingredients of a satisfactory most general and rigorous but practical notion of abstract machines—instruction set machines (abstract dynamics) and structures as states (abstract statics)—were in the literature for more than 15 years, ready to complete the longstanding structural programming endeavour (see [53]) by lifting it from particular machine or programming notation to truly abstract programming on arbitrary structures, nobody performed the natural step to simply combine these two notions and obtain ASMs. Probably this has to do with the fact that in theoretical computer science (and apparently not only there [117]), the value of declarative and of compositional (usually syntax driven) instead of procedural methods has been largely overestimated. In theory circles it still is widely regarded as scientifically not qualifying or rewarding to study the operational, process and machine oriented view of com-

putation[10], with the consequence that the proposed *pure* (functional, algebraic, axiomatic, logical) methods, impractical as they are for dealing with system dynamics, up to now did not really influence or support the way practitioners work. It needed a new start, from scratch (in an attempt to sharpen Turing's thesis) and free from *ideo*-logical a prioris, which eventually led to the right notion, Abstract State Machines, where static and dynamic methods instead of excluding each other can be fruitfully combined.

There is an analogy to the discovery of the notion of Turing machine. During the first third of this century numerous definitions were proposed to formalize the intuitive concept of algorithm and computable function, without really capturing in a convincing way the intuitive notion. The discovery of universal Turing machines [129] provided a justifiably general basis for the theory of algorithms and complexity and laid the conceptual ground for the construction of von Neumann computers. Gurevich's notion of ASM clarifies the relation between different virtual machines and computation models and has laid the ground upon which a practical well founded method could be built for design and analysis of complex real-life hardware/software systems. By the way it has also led to interesting new developments in logic and complexity theory [13, 70, 14].

It would not come as a surprise to see the historian confirm in the retrospective that it was a thorny way which led to ASMs. Büchi, explaining "an approach he developed with Jesse B. Wright in the 1950s in the Logic of Computers Group at the University of Michigan, Ann Arbor"(Sic) (D. Siefkes, op.cit.p.5) speaks about "an intriguing interpretation ... of an algebra as a machine" where "the mathematical structures ... in our theory are to represent the real objects"[49, p.76]—to exploit it for an (elegant) algebraic interpretation of finite automata as finite algebras with unary functions. Scott [118] lifted finite automata to abstract machines by adding, to the change of the internal control state, the inspection and transformation of a memory state, but he formulated these memory state transformations by static functions $f : M \to M$ whereby they remain global and inherit the annoying frame problem, as is the case in numerous followers of this approach [59, 51, 82, 92, 83]. The ASM *machine instructions* (guarded function updates) enable us to distinguish between (and to combine the advantages of) a global state view and local state transformations, a particularly important feature for distributed systems, see below the discussion of the concept of Globally Asynchronous Locally Synchronous (GALS) machines.

3.2 Encompassing Sequential Models of Computation

The VDM approach to system modelling is very carefully explained in [60]. VDM is restricted to sequential runs. The abstraction level is fixed, for sets by the VDM-SL types (which have to be built from basic types by constructors),

[10] Michel Sintzoff, in an e-mail discussion in June 1996, commented: "If the operational style of dynamical systems is "dirty", then Poincaré should be removed from the history of mathematics". See the discussion in [120, section 2.4] of the fundamental role operational methods play for expert knowledge.

for functions it permits explicit and implicit definitions, for operations one is allowed to use procedures (with possible side effects). The notion of state is restricted to records of variables (sets of 0-ary functions) which are classified into read/write variables only. The tutorial [60] is biased towards functional modelling[11] although assignments and state based specifications are supported by VDM. It would be interesting to know whether and to what extent the IFAD tool support for VDM could be enriched to cover the greater semantical freedom offered by ASM modelling, even if restricted to the sequential case.

Abrial's B-Method is the method which in spirit and conceptually comes closest to the ASM method. The B-method is model oriented, the way also its predecessors Z and VDM are, but in addition it is based on Abstract Machine (instead of purely axiomatic or functional) Notation (AMN). A difference in spirit comes out in the way abstract machine programs are related to the justification that the program does what it is supposed to do. In B "the idea is to accompany the technical process of *program* construction by a similar process of *proof* construction, which guarantees that the proposed program agrees with its intended meaning" and as a consequence the "simultaneous concerns about the architecture of a program and that of its proof" [3, p.XI] characterize the B-method. They determine its compositional constructs (guards, sequentialization, parallelism, non-deterministic choice of actions or objects, inclusion/import, using/seeing) as well as its refinement concept and thereby guide the specification and design process. This is enforced by the supporting proof tools. The ASM designer has a greater freedom to choose, from case to case and depending on the design level, the most appropriate and convenient way to proceed; the method avoids any a priori fixed combination of design language and proof system, inviting the software engineer to use whatever form of mathematical language, programming notation and rigorous reasoning may be useful for defining the desired system, for implementing it through further detailing (see above the comparison of the ASM and B refinement concepts) and for making one "convinced that the software system in question is indeed correct" [3, p.XV]. ASMs adhere to the idea that there are many layers not only for the design, but also for justifying a design (i.e. of correctness "proofs").

A conceptual difference between the B-method and the ASM method stems from the different epistemological explanation of what "is" an abstract machine. For Abrial it "is a concept that is very close to certain notions well-known in programming, under the names of modules, classes or abstract data types" [3, p.XV] and as a matter of fact is defined on the basis of pre- and post-conditions and specifically of Dijkstra's weakest pre-condition theory. ASMs are the practical hardware/software design analogue of Turing machines, result of a modern analysis of Turing's thesis [76]; they are based only on standard mathematical

[11] because this allows to go, as the authors say, "without the distraction of operation syntax, side-effects and access restrictions to external variables" [60, p.XII]. The ASM philosophy is as follows. If there is an effect which plays a role, it should stand out explicitly; if it does not play a role, it should be hidden (abstracted away) so it does not appear not even as side-effect.

logic (which includes Thue's [128] nowadays omnipresent notion of transition system). The natural semantical basis for ASMs protects the notion from certain complications the B-method has to face due to its conceptual heritage. For example the fundamental assignment operation, which *is* used throughout, is not introduced as semantically basic but has to be defined syntactically through the not completely elementary concept of substitution. The proof method is tailored for termination proofs, using specific refinement techniques which in certain situations lead to problems (see [110,7]). The axiomatic foundation by Dijkstra's weakest precondition theory is not easy to understand; furthermore it seems to lead to the tendency to specify machine operations using logical descriptions instead of expressing the dynamics directly through local state updates. AMN deals only with sequential machines (although Abrial is working on an extension of AMN to distributed computations).

There are also some technical differences. The state notion in Abrial's AMN is that of a set of finitely many variables, instead of arbitrary Tarski structures. Sets and functions are supposed to be finite (whereas with ASMs dealing with the finiteness problem can be postponed to the moment when it comes to implement the abstract machine into specific control and data structures and to worry about garbage collection). Anchoring ASMs in classical logic avoids the well definedness problem without having to rely upon neither three-valued logic nor any special proof calculus to circumvent the logic of partial functions [12].

Eilenberg's X-machines [59] are state machines which, in their current *control state*, modify their current *memory state* as indicated by their *input*, produce an *output* and pass to the next control state; differently from a finite automaton, an X-machine has as input σ not a letter, but a partial memory function $\sigma : X \to X$ which is consumed by applying it to the current memory state. X-machines are a special form of the abstract machines defined by Scott in 1967 [118]. In 1988 Holcombe used X-machines for describing Turing machines, Petri nets, finite automata and a small storage system and proposed "to integrate the machine ideas central to the modelling of the control of the processing together with the data type descriptions of the processing operations in a unified methodology ... for dynamic system specification"[82, p.72]. Starting with [92] a practically important subclass of these machines, called stream X-machines, has been used to generalize Chow's finite state machine testing method [50,83].

X-machines are specializations of what we call *control state ASMs*, i.e. ASMs where all the rules have the (usually graphically displayed) form

$$s \xrightarrow[Rule]{} s'$$

with elements s, s' of a set of control states (the "internal states" in finite automata) and standing for

$$\textbf{if } currstate = s \textbf{ then } \begin{array}{l} currstate := s' \\ Rule \end{array}$$

X-machines are sequential control state ASMs where *Rule* is a single update by a global memory function f, i.e. of form $currmemory := f(currmemory)$.

Stream X-machines are a specialization of what we call *Mealy-ASMs*, i.e. ASMs with a monitored input function *in* where all the rules have the (usually graphically displayed) form

$$s \xrightarrow[Rule]{Cond(in)} s'$$

standing for

$$\textbf{if } currstate = s \wedge Cond(in) \textbf{ then } \underset{Rule}{currstate := s'}$$

Rule may and usually does include an output update $out := \ldots$. Mealy-ASMs provide a uniform semantical basis for event driven machine computation models, like Message Sequence Charts, Discrete Event Systems [112], etc.. Stream X-machines are Mealy-ASMs where *Rule* is restricted as in X-machines, the output is defined as concatenation of the current with the previous output, and *Cond(in)* is restricted to $\sigma \in A$ for the underlying function alphabet A.

The (stream) X-machines, in the form they are defined and used in [82, 92, 83], are exposed to the difficulties of the frame problem. They inherit from their algebraic origin a global (Cartesian product) memory view, showing up in the frequent "don't care" and "no change" entries in function definitions. This is different from the combination of global memory view with local update view in control state ASMs, Mealy-ASMs and ASMs in general. Also in ASMs there is a global set X, the so called *superuniverse* [75], which together with the functions defined on X forms the global state (memory and control). But memory changes can nevertheless be thought of as happening locally, by formulating the intended updates $f(t_1, \ldots, t_n) := t$ in rules. The semantical definition of ASMs states once and for all that controlled functions change only due to rule execution, freeing the designer from having to think about this frame condition and having to present particular instances of it at each state change. It would be interesting to know how much of the decomposition, structuring and refinement principles from algebraic automata theory can be extended to capture some of the semantical structuring possibilities for ASM. The same question applies to extensions of Chow's finite automata based testing method [50, 83].

Parnas tables [107] exploit the mathematical matrix notation for a concise geometrical layout of multiple case distinctions in predicate and function definitions. Various semantical interpretations of such tables have been introduced for software documentation purposes, see [1, 88] for a survey and [87] for a formal semantics. We illustrate three frequently used types of Parnas tables by ASM definitions which provide a simple and uniform semantical basis for such 2-dimensional notations.

Normal tables are used to assign a value $t_{i,j}$ to $f(x, y)$ in case the row condition $r_i(x)$ and the column condition $c_j(y)$ are both true. The consistency of such definitions usually results from the disjointness of the row conditions and of the column conditions.

$N(f)$	c_1	...	c_n
r_1	$t_{1,1}$	...	$t_{1,n}$
⋮	⋮		
r_m	$t_{m,1}$	...	$t_{m,n}$

meaning **if** $r_i(x) \wedge c_j(y)$ **where** $1 \leq i \leq m$
then $f(x,y) := t_{i,j}$ $1 \leq j \leq n$

Inverted tables are used to assign a value t_j to $f(x,y)$ in case at least one of the leading conditions $r_i(x,y)$ (which usually depend only on one of the variables) and the corresponding side condition $c_{i,j}(x,y)$ are both true.

$I(f)$	t_1	...	t_n
r_1	$c_{1,1}$	...	$c_{1,n}$
⋮	⋮		
r_m	$c_{m,1}$	...	$c_{m,n}$

meaning **if** $r_i(x,y) \wedge c_{i,j}(x,y)$ **where** $1 \leq i \leq m$
then $f(x,y) := t_j$ $1 \leq j \leq n$

Decision tables are used to trigger an action t_j (read: to assign a value t_j to $f(x_1,\ldots,x_k)$) in case all the conditions $r_{i,j}(x_1,\ldots,x_k)$ appearing in column j are satisfied, where usually each condition $r_{i,j}(x_1,\ldots,x_k)$ concerns one term sequence s_i which guides the case distinction and is displayed in the leading column.

$D(f)$	t_1	...	t_n
s_1	$r_{1,1}$	...	$r_{1,n}$
⋮	⋮		
s_m	$r_{m,1}$	...	$r_{m,n}$

meaning **if** $\bigwedge_{1 \leq i \leq m} r_{i,j}(s_i)$ **where** $1 \leq j \leq n$
then $f(x_1,\ldots,x_k) := t_j$

The theory for Parnas tables and the underlying Four Variable Model [107] is focussed on sequential systems with states consisting of finitely many variables, classified into monitored and controlled; recently also shared variables have been included [104]. The state variables are treated as function of time and thereby capture system dynamics; this does not work however for distributed features which are not only a function of time. In applications of Parnas tables we observe a frequent use of "auxiliary" functions, predicates, data structures, etc. whose semantical integration is taken for granted. The ASM definition of tables provides a simple rigorous foundation for this natural form of semantical table modularization and decomposition, in particular through the fine grained function classification explained above; it also makes the ASM refinement techniques applicable to tables and thus helps to document the system decomposition. The ASM definition of tables also overcomes the limitations of the 2-dimensional geometrical layout due to which the tables typically define functions and predicates which are viewed as monadic or binary (although in another analysis layer, each argument may appear as a tuple). Despite their state based dynamic orientation, Parnas tables are reminiscent of a "declarative" view of computation: the 1-step transition relation is expressed not by transition rules using destructive updates (assignments), but by mathematical formulae which use the well known x/x'-notation of temporal logic and thereby inherit the frame problem—appearing in the so-called NC-conditions ("No Change") of tables.

3.3 Encompassing Distributed Models of Computation

Petri nets are among the most popular models for distributed computation. We do not want to go here into the details of the numerous different versions of Petri nets. Instead we give a definition of naturally *generalized* Petri nets which covers all cases in the literature. Any first-order condition (on markings, colours, whatever you want) is allowed as precondition, involving not only single but arbitrary combinations of places; any function update (to change the marking, the colours, whatever) is allowed for any combination of places affected by a rule. This generalization supports the integration of Petri nets into the ASM framework and vice versa. Here is the definition.

Let P be a set (of so called *places*). A Petri-ASM (over P) is a multi-agent ASM where the rules of each agent are of the following (usually graphically displayed) form:

$$\begin{aligned} \textbf{if } \alpha_1(p_1) \wedge \ldots \wedge \alpha_n(p_n) \textbf{ then } & f_1(q_1) := t_1 \\ & \vdots \\ & f_m(q_m) := t_m \end{aligned}$$

where p_i, q_j are finite sequences of elements of P, α_i are arbitrary first-order formulae, f_j are arbitrary functions on places and t_j are arbitrary terms used to determine values to be associated to functions at places. In Petri nets typically the p_i, q_j are sequences of length 1 and some p_i are idential to some q_j. In the case of standard marked Petri nets for example, $\alpha_i \equiv L_i \subseteq m(p_i)$ where $m : P \to M$ is the dynamic marking function and L_i is the static marking precondition (so called label) attached by the rule to p_i; f_j describes the intended update of $m(q_j)$ (deleting the precondition marks from q_j, if there are any, and adding the postcondition marks R_j attached by the rule to q_j, say $m(q_j) := m(q_j) - L_j + R_j$). By deciding which sequential agent gets which rules to execute, the intended distributed nature of Petri net computations is realized faithfully. Attributing transitions to agents can simplify the analysis of runs and the corresponding proofs. Specializations of Petri nets by particular synchronization mechanisms are reflected by corresponding (order) restrictions on the runs of the Petri-ASM.

Last but not least we want to mention the very interesting concept of **Codesign Finite State Machines** (CFSMs), defined with the goal to "combine the advantages of verifiability in synchrony and flexibility in asynchrony in a globally asynchronous, locally synchronous (GALS) model" [91, section 4.1] and synthesizing a detailed analysis of the advantages and drawbacks of the major models of computation in use for system design. CFSMs can be defined as distributed ASMs whose components are Mealy-ASMs, possibly coming together with a global scheduler controlling the interaction of the components and/or with timing conditions in case of durative instead of atomic component actions. The (in ASM terminology sequential) Mealy-ASM components have a locally synchronous behavior, whereas the partial order of the distributed ASM reflects the globally asynchronous system character. It should be stressed that also the data structure capabilities of Mealy-ASMs are exploited for the definition of

CFSMs; namely the transitions are allowed to refer to arbitrary combinational (external and instantaneous) functions. See [91] for further details.

4 Conclusion and Future Research

Various prototypical tools have been developed for designing and executing ASMs; for an overview see [54] in this volume. More advanced and industrially satisfactory tool support is needed for defining, simulating and visualizing, debugging, transforming (refining, implementing, where possible through code generation), analysing (testing and verifying) ASMs. The tool environment should support to capture design knowledge in a rigorous and electronically available way. It should be linked as much as possible to established design flows and exploit their achievements; for an encouraging successful project in this context see the description [89] in this volume. Together with an advanced tool environment a model and a proof theory of ASMs are needed which in particular define practical refinement principles, ideally together with corresponding proof schemes; see [115] for a good start. The proof theory we need should alleviate the formalization effort encountered in practical applications, instead of making things more difficult; it should support abstraction and refinement, structuring and layering in proofs instead of focussing on a single (worse if only machine oriented) level of detailing. The ASM theory should build upon and extend what has been achieved in more specific design approaches which are supported by a rich tool environment, like CFSMs, B, UML, VDM, Petri nets, etc. This is a ridge walk between freedom and discipline, creativity and pattern oriented design, generality and specialization, expressability and limitations by tool support.

The "codeless form of programming" offered by ASMs helps porting application programs from one platform or language to another and could lead to fruitful applications for plug-and-play software technology [9]. Paradigmatic and parameterized ASM components and (de)composition techniques for constructing them have to be defined and to be made available in libraries. ASMs should also be put to use to enhance current (mostly signature oriented) software architecture description techniques by adding semantical content to the structural definitions. This is particularly promising at the levels of what in [122] is called *conceptual architecture* and *module interconnection architecture*. Conceptual architecture refers to the ground model level where domain specifics play a major role; the module interconnection architecture level reflects implementation decisions which are independent of any particular progamming language, such as the logical/functional decomposition, layers with allowable import/export relations, interface constraints, etc.. The integration potential of ASMs, as a universal model of computation, is crucial in achieving the goal to capture the overall behavior of a complex system in terms of the behavior of the interacting components, by whatever rigorous descriptions are appropriate (static, dynamic, functional, state-based, etc.). This will not only improve the reuse and

the reconfiguration possibilities for system descriptions, but also contribute to a satisfactory comparative analysis of different architectural models.

It would be interesting to investigate the possibilities ASMs offer for a theoretical foundation of good testing methods for high-level design. ASMs can help to solve the crucial and essentially creative part of test case selection, given that this selection is driven by appplication domain expert knowledge and thus can be formulated using the ASM ground model. The ground model may allow one to even generate a test scheme. Furthermore the ground model supports solving the oracle problem of testing: the expected output, which has to be compared with the execution output, can be defined using the ground model specification (which is independent of the programming language where the system will be encoded). A similar remark applies to static testing (code inspection) where one has to formulate the properties to be checked.

No doubt in this presentation ASMs have received much praise. To figure out whether this is only adulation or whether there is a *fundamentum in re*, there is only one thing one can do: put ASMs to the test.

Acknowledgment. Thanks for the critical remarks received on drafts of this paper from Donatella Barnocchi, Don Batory, Jonathan Bowen, Axel Dold, Mike Holcombe, Jim Huggins, Hans Langmaack, Hanno Nickau, Peter Päppinghaus, Gerhard Schellhorn, Peter Schmitt, Wolfgang Schönfeld, Dilip Soni, Lothar Thiele, Kirsten Winter, Jeff Zucker. To Michel Sintzoff thanks also for the stimulating discussion on the history of the concept of abstract machines. Last but not least thanks to my students in the Fall 1998 *Specification Methods* Course, in particular Marenco Yari, Fabrizio Sanna, Cristian Gozzi, Tommaso Mangini, Emanuele Lupi, Andrea Pieroni and Cinzia Ridolfo, for the ASM reuse experiments mentioned above.

References

1. Abraham, R.: Evaluating Generalized Tabular Expressions in Software Documentation. M. Eng. Thesis, CRL Report 346, McMaster University, Hamilton, Ontario, Canada (1997)
2. http://www.eecs.umich.edu/gasm/, http://www.uni-paderborn.de/cs/asm.html
3. Abrial, J.-R.: The B-Book. Assigning Programs to Meanings. Cambridge University Press (1996)
4. Anlauff, M., Kutter, P., Pierantonio, A.: Formal Aspects of and Development Environments for Montages. In: Sellink, M. (ed): 2nd International Workshop on the Theory and Practice of Algebraic Specifications. Springer Workshops in Computing (1997)
5. Araujo, L.: Correctness Proof of a Distributed Implementation of Prolog by Means of ASMs. J. of Universal Computer Science. Special ASM Issue 3(5) (1997)
6. Aristotle. Analytica Posteriora I,1, 71a,1 sq.
7. Banach, R., Poppleton, M.: Retrenchment: An Engineering Variation on Refinement. In: Bert, D. (ed): B'98: Recent Advances in the Development and Use of the B Method. Lecture Notes in Computer Science, Vol. 1393. Springer-Verlag, Berlin Heidelberg New York (1998) 129-147

8. Barnocchi, D.: L"Evidenza" nell'assiomatica aristotelica. Proteus II,5 (1971) 133–144
9. Batory, D., Singhai, V., Sirkin, M., Thomas, J.: Scalable Software Libraries. ACM SIGSOFT'93: Symposium on the Foundations of Software Engineering. Los Angeles/California (1993)
10. Batory, D., Coglianese, L., Goodwin, M., Shafer, S.: Creating Reference Architectures: An Example from Avionics. Symposium on Software Reusability. Seattle/ Washigton (1995)
11. Batory, D., O'Malley, S. : The Design and Implementation of Hierarchical Software Systems with Reusable Components. ACM Transactions on Software Eng. and Methodology (October 1992)
12. Behm, P. Burdy, L., Meynadier, J.-M.: Well Defined B. In: Bert, D. (ed): B'98: Recent Advances in the Development and Use of the B Method. Lecture Notes in Computer Science, Vol. 1393. Springer-Verlag, Berlin Heidelberg New York (1998) 29-45
13. Blass, A., Gurevich, Y.: The Linear Time Hierarchy Theorems for Abstract State Machines. J. of Universal Computer Science. Special ASM Issue, 3(4) (1997) 247-278
14. Blass, A., Gurevich, Y., Shelah, S. : Choiceless Polynomial Time. EECS Dept. University of Michigan, Technical Report CSE-TR-338-97 (1997)
15. Beierle, Ch., Börger, E.: Specification and Correctness Proof of a WAM Extension with Abstract Type Constraints. Formal Aspects of Computing 8(4) (1996) 428-462
16. Beierle, Ch., Börger, E.: Refinement of a Typed WAM Extension by Polymorphic Order-Sorted Types. Formal Aspects of Computing 8(5) (1996) 539-564
17. Beierle, Ch., Börger, E., Đurđanović I., Glässer, U., Riccobene,E.: Refining Abstract Machine Specifications of the Steam Boiler Control to Well Documented Executable Code. In: Abrial, J.-R., Börger, E., Langmaack, H. (eds.): Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control. Lecture Notes in Computer Science, State-of-the-Art Survey, Vol. 1165. Springer-Verlag, Berlin Heidelberg New York (1996) 52–78
18. Börger, E.: A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control. CSL'89. Lecture Notes in Computer Science, Vol. 440. Springer-Verlag, Berlin Heidelberg New York (1990) 36–64
19. Börger, E.: A Logical Operational Semantics for Full Prolog. Part II: Built-in Predicates for Database Manipulations. In: Rovan, B. (ed): MFCS'90. Mathematical Foundations of Computer Science. Lecture Notes in Computer Science, Vol. 452. Springer-Verlag, Berlin Heidelberg New York (1990) 1–14
20. Börger, E.: Logic Programming: The Evolving Algebra Approach. In: Pehrson, B., Simon, I. (eds): IFIP 13th World Computer Congress 1994. Volume I: Technology and Foundations. Elsevier, Amsterdam (1994) 391–395
21. Börger, E.: Why Use Evolving Algebras for Hardware and Software Engineering. In: Bartosek, M., Staudek, J., Wiedermann, J.(eds): SOFSEM'95. 22nd Seminar on Current Trends in Theory and Practice of Informatics. Lecture Notes in Computer Science, Vol. 1012. Springer-Verlag, Berlin Heidelberg New York (1995)236–271
22. Börger, E., Dässler, K.: Prolog: DIN Papers for Discussion. ISO/IEC JTCI SC22 WG17 Prolog standardization document, no. 58,.NPl, Middlesex (1990) 92–114
23. ISO/IEC 13211-1 Information Technology-Programming Languages-Prolog-Part 1: General Core (1995)
24. Börger, E., Del Castillo, G.: A Formal Method for Provably Correct Composition of a Real-Life Processor out of Basic Components (The APE100 Reverse Engineering Project). Proc. of the First IEEE International Conference on Engineering

of Complex Computer Systems (ICECCS'95). Extended version in: Gurevich, Y., Börger, E. (eds): Evolving Algebras. Mini-Course. University of Aarhus. BRICS NS-95-4 (1995) 195–222
25. Börger, E., Demoen, B.: A Framework to Specify Database Update Views for Prolog. In: Maluszynski, M. J. (ed): PLILP'91. Lecture Notes in Computer Science, Vol. 528. Springer-Verlag, Berlin Heidelberg New York (1991) 147–158. See also: The View on Database Updates in Standard Prolog: a Proposal and a Rationale. In: ISO/IEC JTC1 SC22 WG17 Prolog Standardization Report no.74 (February 1991) pp. 3-10
26. Börger, E., Đurđanović, I.: Correctness of Compiling Occam to Transputer Code. Computer Journal 39(1) (1996) 52–92
27. Börger, E., Đurđanović, I., Rosenzweig, D.: Occam: Specification and Compiler Correctness. Part I: The Primary Model. In: Olderog, E.-R. (ed): Proc. of PROCOMET'94 (IFIP Working Conference on Programming Concepts, Methods and Calculi). North-Holland (1994) 489–508
28. Börger, E., Glässer, U.: A Formal Specification of the PVM Architecture. In: Pehrson, B., Simon, I. (eds): IFIP 13th World Computer Congress 1994. Volume I: Technology and Foundations. Elsevier, Amsterdam (1994) 402–409
29. Börger, E., Glässer, U.: Modelling and Analysis of Distributed and Reactive Systems Using Evolving Algebras. In: Gurevich, Y., Börger, E. (eds): Evolving Algebras. Mini-Course. University of Aarhus. BRICS NS-95-4 (1995) 128–153
30. Börger, E., Glässer, U., Mueller, W.: The Semantics of Behavioral VHDL'93 Descriptions. EURO-DAC'94 European Design Automation Conference with EURO-VHDL'94. Proc. IEEE CS Press, Los Alamitos/CA (1994) 500–505
31. Börger, E., Glässer, U., Mueller, W.: Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines. In:Delgado Kloos, C., Breuer, P.T. (eds): Semantics of VHDL. Kluwer (1995) 107–139
32. Börger, E., Gurevich, E., Y.., Rosenzweig, D.: The Bakery Algorithm: Yet Another Specification and Verification. In: Börger, E. (ed): Specification and Validation Methods. Oxford University Press, (1995) 231–243
33. Börger, E., Huggins, J.: Annotated Bibliography on Abstract State Machines (ASMs). EATCS Bulletin (February 1998)
34. Börger, E., Lopez-Fraguas, F.J., Rodrigues-Artalejo, M.: A Model for Mathematical Analysis of Functional Logic Programs and their Implementations. In: Pehrson, B., Simon, I. (eds): IFIP 13th World Computer Congress 1994. Volume I: Technology and Foundations. Elsevier, Amsterdam (1994) 410–415. Full version: Towards a Mathematical Specification of Narrowing Machines. Report DIA 94/5, Dep. Informática y Automática. Universidad Complutense, Madrid (March 1994) 1–30
35. Börger, E., Mazzanti. , S.: A Practical Method for Rigorously Controllable Hardware Design. In: Bowen, J.P., Hinchey, M.B., Till, D. (eds): ZUM'97: The Z Formal Specification Notation. Lecture Notes in Computer Science, Vol. 1212. Springer-Verlag, Berlin Heidelberg New York (1997) 151–187
36. Börger, E., Mearelli, L.: Integrating ASMs into the Software Development Life Cycle. J. of Universal Computer Science, Special ASM Issue, 3 (5) (1997) 603-665
37. Börger, E., Riccobene, E.: A Formal Specification of Parlog. In: Droste, M., Gurevich, Y. (eds): Semantics of Programming Languages and Model Theory. Gordon and Breach (1993) 1–42
38. Börger, E., Rosenzweig, D.: An Analysis of Prolog Database Views and their Uniform Implementation. In: Prolog. Paris Papers–2. ISO/IEC JTC1 SC22 WG17 Prolog Standardization Report no.80 (July 1991) 87–130

39. Börger, E., Rosenzweig, D.: A Mathematical Definition of Full Prolog. Science of Computer Programming. 24 (1995) 249–286
40. Börger, E., Rosenzweig, D.: The WAM–Definition and Compiler Correctness. In: Beierle, Ch.,Plümer, L. (eds): Logic Programming: Formal Methods and Practical Applications. Elsevier Science B.V./North–Holland (1995) 20–90
41. Börger, E., Salamone, R.: CLAM Specification for Provably Correct Compilation of CLP($\mathcal{R}$) Programs. In: Börger, E. (ed): Specification and Validation Methods. Oxford University Press, (1995) 97–130
42. Börger, E., Schmitt, P.: A Formal Operational Semantics for Languages of Type Prolog III. Lecture Notes in Computer Science, Vol. 533. Springer-Verlag, Berlin Heidelberg New York (199) 67–79
43. Börger, E., Schulte, W.: Programmer Friendly Modular Definition of the Semantics of Java. In: Alves-Foss, J. (ed): Formal Syntax and Semantics of Java. Lecture Notes in Computer Science, Vol. 1523. Springer-Verlag, Berlin Heidelberg New York (1999) 353 - 404. Extended Abstract in: Berghammer, R., Simon, F. (eds): Programming Languages and Fundamentals of Programming. University of Kiel (Germany) TR 9717 (1997) 175–181.
44. Börger, E., Schulte, W.: Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In: Brim, L., Gruska, J., Zlatuska, J. (eds): Proc. MFCS'98. Lecture Notes in Computer Science, Vol. 1450. Springer-Verlag, Berlin Heidelberg New York (1998) 17–35
45. Börger, E., Schulte, W.: Initialization Problems for Java. Software—Concepts and Tools 20 (4) (1999)
46. Börger, E., Schulte, W.: Modular Design for the Java Virtual Machine Architecture. In: Börger, E. (ed): Architecture Design and Validation Methods. Springer Verlag, Berlin Heidelberg New York 1999
47. Bowen, D. Personal communication. Palo Alto (5.11.1991)
48. Bowen, J.P.: Formal Specification and Documentation Using Z: A Case Study Approach. Int. Thomson Computer Press (1996)
49. Büchi, J.R.: Finite Automata, their Algebras and Grammars. Siefkes, D. (ed). Springer-Verlag (1988)
50. Chow, T.S.: Testing Software Design Modeled by Finite State Machines. IEEE Trans.Softw.Engineering 4(3) (1978) 178–187
51. Cremers, A.B.C., Hibbard, T.N.: Formal Modeling of Virtual Machines. IEEE Transactions on Software Engineering SE-4(5) (1987) 426–436
52. Dahl, O.: Discrete Event Simulation Languages. In: F. Genuys (ed): Programming Languages. Academic Press (1968) 349–395
53. Dahl, O., Dijkstra, E.,Hoare, C.: Structured Programming. Academic Press (1972)
54. Del Castillo, G.: Towards Comprehensive Tool Support for Abstract State Machines: The ASM Workbench Tool Environment and Architecture. This volume
55. Del Castillo, G., Hardt, W.: Fast Dynamic Analysis of Complex HW/SW-Systems based on Abstract State Machines. IEEE Proc. 6th. International Workshop on HW/SW Co-Design (CODES/CASHE'98). Washington (March 1998)
56. Dijkstra, E.W.: Structure of the T.H.E. Multiprogrammming System. Communications of ACM 11 (1968) 341–346
57. Dijkstra, E.W.: Notes on Structured Programming. In: Structured Programming. Academic Press, New York (1972) 1–82
58. Dold, A.: A Formal Representation of Abstract State Machines using PVS. Verifix Report Ulm/6.2 (July 1998)1–25
59. Eilenberg, S.: Automata, Languages and Machines. Vol.A. Academic Press (1974)

60. Fitzgerald, J., Gorm Larsen, P.: Modelling Systems. Practical Tools and Techniques in Software Development. Cambridge University Press (1998)
61. Fuchs, N.E.: Specifications are (Preferably) Executable.IEE/BCS Software Engineering Journal 7(5) (1992) 323–334
62. Gaudel, M.C.: Génération et Preuve de Compilateurs Basées sur une Sémantique Formelle des Langages de Programmation. Thèse, L'Institut National Polytechnique de Lorraine (1980)
63. Geist, A., Beguelin, A., Dongarra, J.,Jiang, W., Manchek, B., Sunderam, V.: PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187. Oak Ridge National Laboratory, Oak Ridge/Tennessee 37831 (September 1994)
64. Glässer, U., Gotzhein, R.: Towards a New Formal SDL Semantics - Outline of an ASM Behavior Model. Submitted to 9th SDL Forum, Montreal/Quebec (21-25 June 1999) http://www.iro.umontreal.ca/SDL/
65. Glässer, U., Karges, R.: Abstract State Machines Semantics of SDL. J. of Universal Computer Science 3 (12) (1997) 1382–1414
66. Glässer, U., Prinz, A.: Abstract State Machines Semantics of SDL. Submitted (1999)
67. Giese, M., Kempe, D., Schönegge, A.: KIV zur Verifikation von ASM-Spezifikationen am Beispiel der DLX-Pipelining Architektur. University of Karlsruhe, Institute for Logic, Complexityy and Deduction Systems. Int. Rep. 16/97 (1997) 1–37
68. Goerigk, W., Hoffmann, U.: Rigorous Compiler Implementation Correctness: How to Prove the Real Thing Correct. This volume
69. Gosling, J., Joy, B., Steele, G.: The Java(tm) Language Specification. Addison Welsley (1996)
70. Grädel, E., Gurevich, Y.: Metafinite Model Theory. Lecture Notes in Computer Science, Vol. 960. Springer-Verlag, Berlin Heidelberg New York (1995) 313–366
71. Gurevich, Y: A New Thesis. Abstracts. American Mathematical Society (August 1985) 85T-68-203.
72. Gurevich, Y.: Algorithms in the World of Bounded Resources. In: Herken, R. (ed): The Universal Turing Machine–A Half-Century Story. Oxford University Press(1988) 407–416.
73. Gurevich, Y.: Logic and the Challenge of Computer Science. In: Börger, E. (ed): Current Trends in Theoretical Computer Science. Computer Science Press (1988) 1–57
74. Gurevich, Y.: Evolving Algebras: An Attempt to Discover Semantics. Bulletin EATCS 43 (1991) 264–284. Slightly revised in: Rozenberg, G., Salomaa, A. (eds): Current Trends in Theoretical Computer Science. World Scientific (1993) 274–308
75. Gurevich, Y.: Evolving Algebra 1993: Lipari Guide. In: Börger, E. (ed): Specification and Validation Methods. Oxford University Press (1995) 9–36
76. Gurevich, Y.: The Sequential ASM Thesis. Bulletin of the EATCS (February 1999)
77. Hayes, I.J., Jones, C.B.: Specifications are not (Necessarily) Executable. IEE/BCS Software Engineering Journal 4(6) (1989) 330–338
78. Heberle, A., Löwe, W., Trapp, M.: Safe Reuse of Source to Intermediate Language Compilations. In: Chillarege, R. (ed): Proc. 9th. Int. Symp. on Software Reliability Engineering (1998) http://www.chillarege.com/issre/fastabstracts/98417.html
79. Hinrichsen, H.: Formally Correct Construction of a Pipelined DLX Architecture. Darmstad University of Technology, Dept. of Electrical and Computer Engineering. TR 98-5-1 (1998)
80. Hoare, C.A.R.: The Structure of an Operating System. Lecture Notes in Computer Science, Vol. 46. Springer-Verlag, Berlin Heidelberg New York (1976) 242–265

81. Hoare, C.A.R.: Mathematical Models for Computing Science. Manuscript (August 1994) 1–65
82. Holcombe, M.: X-Machines as a Basis for Dynamic System Specification. Software Engineering Journal 3(2) (1988) 69–76
83. Holcombe, M., Ipate, F.: Correct Systems. Springer-Verlag, Berlin Heidelberg New York (1998)
84. Horning, J., Randell, B.: Process Structuring. Computing Surveys 5 (1973) 5–30
85. Huggins, J. K., Van Campenhout, D.: Specification and Verification of Pipelining in the ARM2 RISC Microprocessor. ACM Transactions on Design Automation of Electronic Systems 3 (4) (October 1998)
86. IEEE Standard VHDL Language Reference Manual—IEEE Std 1076–1993. The Institute of Electrical and Electronics Engineering. New York/NY (1994)
87. Janicki, R.: On a Formal Semantics of Tabular Expressions. Faculty of Engineering, McMaster University. Hamilton/Ontario. CRL Report 355 (1997). Short version in: Towards a Formal Semantics of Parnas Tables. ACM/IEEE Proceedings of the 17th International Conference on Software Engineering, Seattle (April 1995) 231–240
88. Janicki, R., Parnas, D.L., Zucker, J.I.: Tabular Representations in Relational Documents. In: Brink, C., Kahl, W., Schmidt, G.(eds): Relational Methods in Computer Science. Springer-Verlag, Berlin Heidelberg New York (1997) 184–196
89. Kutter, P.W., Schweitzer, D., Thiele, L.: Integrating Formal Domain Specific Language Design in the Software Life Cycle. This volume
90. Kwon, K.: A Structured Presentation of a Closure-Based Compilation Method for a Scoping Notion in Logic Programming. J. of Universal Computer Science, Special ASM Issue, 3(4)(1997) 341–376
91. Lavagno, L., Sangiovanni-Vincentelli, A., Sentovich, E.M.: Models of Computation for System Design. In: Börger, E. (ed): Architecture Design and Validation Methods. Springer- Verlag, Berlin Heidelberg New York (1999)
92. Laycock, G.T.: The Theory and Practice of Specification Based Testing. PH.D. Thesis, University of Sheffield, UK (1992)
93. Lötzbeyer, A.: Task Description of a Fault-Tolerant Production Cell. FZI, University of Karlsruhe/Germany. Version 1.6 (June 17, 1996)
94. Lötzbeyer, A., Mühlfeld, R.: Task Description of a Flexible Production Cell with Real Time Properties. FZI, University of Karlsruhe and Siemens ZT SE 1/Germany. Version 2.1 (July 30, 1996)
95. Mearelli,L.: Refining an ASM Specification of the Production Cell to C^{++} Code. J. of Universal Computer Science, Special ASM Issue 3(5) (1997) 666–688
96. Marcotty, M., Ledgard, H.F.: The World of Programming Languages. Springer-Verlag, Berlin Heidelberg New York (1986)
97. Van der Meulen, M., Clement, T.: Formal Methods in the Specification of the Emergency Closing System of the Eastern Scheldt Storm Surge Barrier. This volume
98. Mosses, P.D.: Denotational Semantics. In: Van Leeuwen, J.(ed.): Handbook of TCS. Elsevier (1990)
99. Mosses, P.D.: Action Semantics. Cambridge University Press (1992)
100. Nielson, H.R., Nielson, F.: Semantics with Applications. Wiley (1992)
101. Pair, C.: Types Abstraits et Sémantique Algébrique des Langages de Programmation. Centre de Recherche en Informatique de Nancy. TR 80-R-011 (February/July 1980) 1–46

102. Päppinghaus, P., Börger, E.: Industrial Use of ASMs for System Documentation. In: Jähnichen, S., Loeckx, J., Wirsing, M. (eds): Logic for System Engineering. Dagstuhl Seminar Report 171 (9710) 26
103. Parnas, D.L.: Information Distribution Aspects of Design Methodology. In: Freiman, C.V. (ed): Proc. of IFIP Congress 1971. Volume 1: Foundations and Systems. North-Holland (1972) 339–344
104. Parnas, D.L.: Personal communication (1997)
105. Parnas, D.L.: "Formal Methods" Technology Transfer Will Fail. J. Systems Software 40 (1998) 195–198
106. Parnas, D.L., Darringer, J.: SODAS and a Methodology for System Design. Proc. AFIPS Fall Joint Conf. Vol.31. Academic Press (1967) 449–474
107. Parnas, D.L., Madey, J.: Functional Documents for Computer Systems. Science of Computer Programming 25 (1995) 41–62
108. Pippenger, N.: Pure versus Impure Lisp. ASM Transactions on Programming Languages and Systems 19 (1997) 223–238
109. Popper, K.: Logik der Forschung (1935)
110. Potet, M.-L., Rouzaud, Y.: Composition and Refinement in the B-Method. In: Bert, D. (ed): B'98: Recent Advances in the Development and Use of the B Method. Lecture Notes in Computer Science, Vol. 1393. Springer-Verlag, Berlin Heidelberg New York (1998) 46-65
111. Pusch, C.: Verification of Compiler Correctness for the WAM. In: Von Wright, J., Grundy, J., Harrison, J.(eds): Theorem Proving in Higher Order Logics (TPHOLs'96). Lecture Notes in Computer Science, Vol. 1125. Springer-Verlag, Berlin Heidelberg New York (1996) 347–362
112. Ramadge, P.J.G., Wonham, W.M.: The Control of Discrete Event Systems. Proc. of the IEEE 77(1) (1989) 81–98
113. Reisig, W.: Elements of Distributed Algorithms. Modeling and Analysis with Petri Nets. Springer-Verlag, Berlin Heidelberg New York (1998)
114. Rischel, H., Sun, H.: Design and Prototyping of Real-Time Systems using CSP and CML. 9th Euromicro Workshop on Teal-Time Systems. Toledo (June 11-13, 1997)
115. Schellhorn, G.: Verifikation abstrakter Zustandsmaschinen. PhD Thesis. University of Ulm (1999)
116. Schellhorn, G., Ahrendt, W.: Reasoning about Abstract State Machines: The WAM Case Study. J. of Universal Computer Science. Special ASM Issue, 3(4) (1997) 377–413
117. Schwank, I.: Zur Konzeption prädikativer versus funktionaler kognitiver Strukturen und ihrer Anwendung. Zentralblatt für Didaktik der Mathematik 6 (1996) 168–183
118. Scott, D.: Some Definitional Suggestions for Automata Theory. J. of Computer and System Sciences 1 (1967) 187–212
119. Shaw, M.: The Impact of Abstraction Concerns on Modern Programming Languages. Proc. IEEE 68 (9) (1980)
120. Shaw, M., Garlan, D.: Formulations and Formalisms in Software Architecture. In: van Leeuwen, J. (ed): Computer Science Today: Recent Trends and Developments. Springer-Verlag, Berlin Heidelberg New York (1995) 307–323
121. Sommerville, I.: Software Engineering. Addison-Wesley (1992)
122. Soni, D., North, R.L., Hofmeister, C.: Software Architecture in Industrial Applications. Proc. 17th ACM Conf. Sw. Engg.. Seattle (1995)
123. Stärk, R.: The Theoretical Foundations of LPTP (A Logic Program Theorem Prover). The Journal of Logic Programming 36 (1998) 241–269

124. Stegmüller, M.M.: Formale Verifikation des DLX RISC-Prozessors: Eine Fallstudie basierend auf abstrakten Zustandsmaschinen. Diploma Thesis, University of Ulm (1998)
125. Stenzel, K., Haneberg, D.: Personal communication. KIV Group, University of Ulm/Germany
126. Tanenbaum, A.S.: Structured Computer Organization (1976)
127. Tarski, A.: Der Wahrheitsbegriff in den formalisierten Sprachen. Studia Philosophica 1 (1936) 261–405
128. Thue, A.: Probleme über Veränderungen von Zeichenreihen nach gegebenen Regeln. Skr.Videnks.Sels I (10)1–34
129. Turing, A.: On computable numbers, with an application to the Entscheidungsproblem. Proc. London Math. Soc. (2) 42 (1937) 230–265
130. UML—Unified Modeling Language. Rational Software Corporation. Updates via the worldwide web (http://www.rational.com)
131. Goerigk, W., Dold, A., Gaul, T., Goos, G., Heberle, A., von Henke, F. W., Hoffmann, U., Langmaack, H., Pfeifer, H., Ruess, H., Zimmermann, W.: Compiler Correctness and Implementation Verification: The *Verifix* Approach. In: Fritzson, P.(ed): Proceedings of the Poster Session of CC'96— International Conference on Compiler Construction. IDA Technical Report LiTH-IDA-R-96-12. Linköping/Sweden (1996)
132. Winter, K.: Model Checking for Abstract State Machines. J. of Universal Computer Science. Special ASM Issue, 3(5) (1997)
133. Wirsing, M.: Handbook of Algebraic Specifications. In: van Leeuwen, J. (ed): Handbook of Theoretical Computer Science B. Elsevier (1990) 675–788
134. Wirth, N.: Program Development by Stepwise Refinement. Comm. ACM 14 (4) (1971)
135. Wirth, N.: Algorithms & Data Structures. Prentice-Hall (1975)
136. Zimmermann, W., Gaul, T.: On the Construction of Correct Compiler Back-Ends: An ASM Approach. J. of Universal Computer Science. Special ASM Issue, 3(5) (1997) 504–567
137. Zurcher, F.W., Randell, B.: Iterative Multi-Level Modelling–A Methdology for Computer System Design. Proc. IFIP Congress 1968. North-Holland, Amsterdam (1968) 867–871

Enriching the Software Development Process by Formal Methods*

Manfred Broy, Oscar Slotosch

Institut für Informatik, Technische Universität München
80290 München, Germany

Abstract. We describe a software development process designed for an integration and usage of formal methods into practical software process models in a scalable way. Our process model is an extension of the V-model, and allows the specification of critical components and the verification of crucial development steps. For different development stages we suggest user-oriented description techniques, based on a common formal semantic. Furthermore we outline methods for the verification of critical development steps. We illustrate our process by developing a small example with some critical aspects.

1 Introduction

The development of software systems is a difficult and error prone task. This is certainly true if systems get very large and complex. However, this may even be true in cases where small to medium size programs have to be developed that are based on complex algorithms, data structures, or patterns of interaction.

Today software development in practice is almost always done in a non-scientific manner [Hoa96] based on pragmatics and heuristics. The development process is structured into phases like:

- analysis and requirements engineering,
- specification,
- design,
- implementation and testing.

In most cases, large parts of the development work are done informally and the correctness of the developed systems relies mainly on the intuition and experience of the developers received by extensive inspections, testing, and prototyping. The debate is still going on whether there is a cost effective alternative to this heuristic approach to software development including and applying so-called formal methods.

In the early days of computing science there was not even a theoretical alternative. The scientific foundation of programming computer systems were not

* This work was supported by the German Information Security Agency (BSI) within the project Quest.

understood. Programming languages were seen as pragmatic, operational notations to control the machine, used on the basis of an intuitive understanding. For many of the programming concepts a theoretical scientific foundation was not available.

Today the theoretical situation has changed. Over the last 30 years the scientific community of computer scientists has developed a solid foundation of software and systems engineering [BJ95,Bro95]. Today there is nearly no programming construct or development concept for which a scientific basis is not available. Of course, still a lot of notation is used in practice for which such a scientific foundation is not worked out explicitly. But this is not due to the lack of theory but rather it has not been done since those people suggesting the notation to a large extend have not yet recognized the virtue of a scientific foundation. For an improvement of the formal basis of software development, it is therefore highly important to have a clear idea what formal methods are good for and how they are combined with informal ones in a way to get the best benefit out of this combination.

Our paper is organized as follows. After a clarification of the term "formal methods" (Section 2), we describe the role of formal methods in the development process and how formal techniques can be combined pragmatically within formal methods. In Section 3 we start with a short presentation of the V-model, and show how formal methods can be integrated into the development process in a scalable way by using them only for the critical parts and for important development steps. In the rest of this paper we illustrate the development process by means of a small example which is presented in Section 4. Section 5 describes different views of the system during the development process, especially in the requirement, design and implementation phases. Furthermore we present graphical description techniques that can be used in a co-development with formal and conventional methods since all description techniques are based on the same semantic model. In Section 6 we present methods that support the engineer in proving the correctness of development steps between different views, and we analyze tool support for different refinement situations.

2 What is a Formal Method

In spite of the enormous amount of work spent on formal methods the notion of a "formal method" is not always made sufficiently precise [Bro96]. We therefore clarify this to begin with. First of all, let us define what we call a method in computing science.

A *method* consists of description concepts, rules for constructing and relating descriptions, and development strategies explaining how and when to apply the method in a goal directed manner. We therefore refer in the following to three ingredients of a method:

D: description concepts,
R: rules, and
S: strategies.

For each of these ingredients we can ask to which extend they are formal in order to evaluate the degree of formality of the method.

Looking at description techniques D we identify the following aspects:

- syntax,
- semantics, and
- proof rules.

A description technique is *fully formal* if it has a *formal syntax* (including both context free syntax and context correctness), a *formal semantics* (described in a precise mathematical way), and a *logical theory* that supports formal proofs about the descriptions and their properties.

A description technique is called *semiformal* if it has a formal syntax but not a formal reference semantics nor proof rules.

The rules R of a method describe the rules for constructing and relating descriptions correctly. For a formal method R consists of proof and transformation rules for the refinement between descriptions, and of (consistency) rules describing the correct construction of descriptions. For a fully formal method we require that the proof rules are given as a formal calculus or embedded by proof obligations into a common logical theory such as first order predicate logic or higher order logic. The same must hold for the transformation and consistency rules.

Finally, a formal method should formalize its development strategies S. Of course, we cannot expect, in general, that the strategies are formalized to a point where the strategies are given by algorithms, such that they can be carried out completely mathematically. In general, the developer guides and controls the development process. We require, however, that the phases and steps are characterized in a formal way, such that it is made precise when a development process is in fact an instance of the overall development strategy.

3 V-Model

The V-model [BD95] is an ISO standard for structuring the software development process. It is supported by the German Ministry of Defense. It contains many detailed steps, guiding the developer through the software engineering process. The name V-model origins from the main scheme of the model, which is depicted as a V (see Figure 1). The left hand side of the V represents the waterfall model of software engineering from requirements down to the program code. The right hand side describes the quality control activities: testing of the components, integrating them, and testing the whole system. The horizontal (dotted) lines express that the requirements of the system have to match the result of the integration test, the integration of the parts has to compose the system according to the structure fixed in the decomposition during the design.

The model is independent from a specific programming or modelling language, and therefore it is used in many developments projects. It supports the development of components, since the activity of testing the components follows

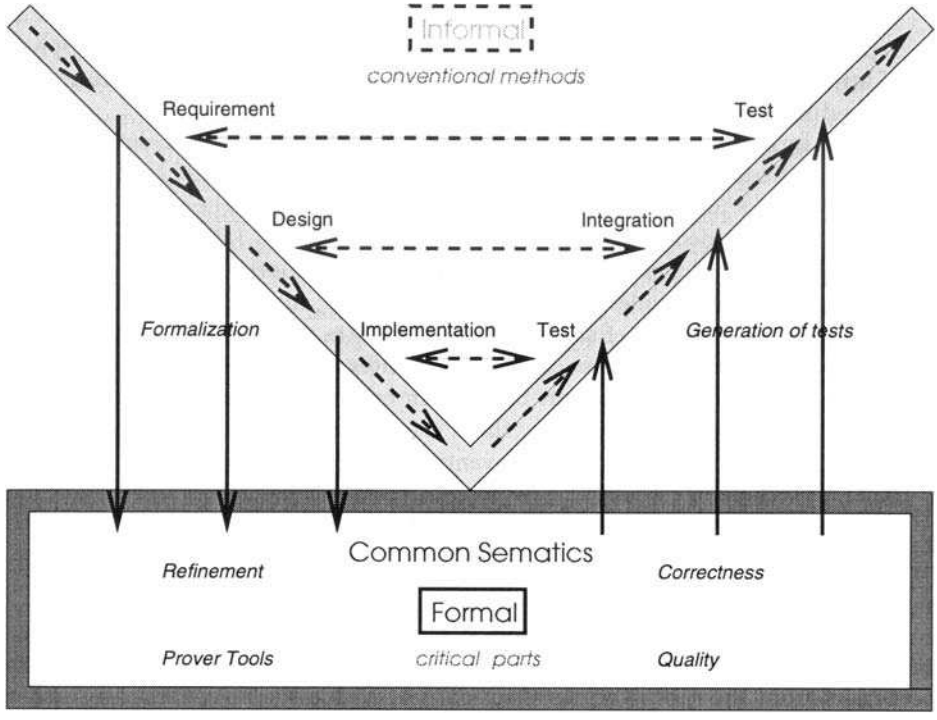


Fig. 1. process model with formal methods

directly after the coding, and errors can be corrected immediately. However, errors made during the decomposition can only be detected during the integration of the system. And errors in the requirements are even more expensive, since they are discovered at the last point, where the whole system is tested.

Formal methods promise that extensive testing is not required if these methods are applied consequently. However, since a complete formal development is not feasible for large projects, the hope for a better process model decreased. We advocate a scalable use of formal methods, by integrating formal methods into the V-model in order to reduce the number of expensive errors and of the test overhead.

Our core concept is to work with user-oriented description techniques, basing on a common formal semantic, to specify critical parts of the system. This basis allows us to ensure the correctness of important development steps, during all phases of the software development process.

There are two degrees in which the conventional development process can be enhanced by formal methods:

- Using formal description techniques in the development process for the modeling of the system views provides these views with a clear denotation and

reduces the number of misunderstandings that occur due to ambiguities of specifications [GSB98]. To facilitate the development process, the applied formal description techniques should be similar to those techniques established in practical methods.

- Using formal refinement to ensure the correctness of critical development steps is the fully formal way for proving the correctness [BJ95,Hin98]. There are two ways in doing this: the *deductive method* allows the developer to do any step, provided that its correctness can be deduced (after the step). The *transformational method* restricts the developer a priori to the set of applicable transformation rules. The first, more flexible, method requires to model the abstract and the refined view and generates proof obligations, whereas the transformational approach requires to integrate the result of the transformations into the development process. In case that the semantics are given by a translation into a calculus, there is a retranslation required to continue with the development on the conventional side. By allowing the developer to prove the correctness of new transformation rules the transformational method can be seen as a special case of the deductive method.

With these different levels of formality, the use of formal methods in the development can be scaled according to the requirements of the project.

We recommend graphical description techniques with formal semantics (see Section 5.5, [Bro95]). The formal semantics allow us to systematically generate test cases from the specifications (Section 6.4, [Sad98,Par95]). This is a further reason for using our formal description techniques in development projects.

4 A small Example: Traffic Lights

We develop the software for a traffic light system that controls a pedestrian crossing. Pedestrians operate two buttons (one at each side of the street) to request the green light. In addition, there are acknowledge lights on both sides of the street to indicate the pedestrians that their request has been excepted by the system.

The requirements for the software to make the system behave like a usual crossing (e.g. both pedestrian and both car lights show consistent signals) are assumed to be realized by the design and the implementation of the system. As part of those functional requirements there are some critical aspects of the system, which have to be guaranteed by the developer:

NO-CRASH: It must be excluded that pedestrians and cars have green lights at the same time.

NO-BLOCK: It must be excluded to block the traffic, i.e. neither pedestrians, nor cars have to wait forever.

SEPARATE: After the red phase for cars there is a yellow signal before the traffic lights are switched to green. After the green phase there is again a yellow light.

In fact, the mentioned requirements are of different nature. The first and the last are are safety conditions, while the second one is a lifeness condition.

5 Formal Description Techniques for System Views

In this section we introduce formal description techniques that occur at different stages during the development process and focus on different views of the system, according to the requirements of the software development process at this stage.

We present graphical description techniques (see [HMR+98]) with common formal semantics for the modelling of the different views. The common formal semantics allows us to define the relation between the different views and the different development stages to support the formal development (see Section 6). The techniques are summarized in Section 5.5 to illustrate the different techniques and to show how the degree of formality can be scaled we use the example of Section 4.

5.1 Requirements Specification

The first views of a system within the development process occur in the requirements specification. The requirements specification covers the following aspects:

- Boundaries of the system to its environment.
- Interactions between the system and its environment (communication channels and messages). The interactions may be incomplete descriptions of the behaviour of the system.
- Critical parts of the system (together with their interactions).

In our example we start with a data model for the messages which we define in a textual notation:

```
data PedLight = PRed | PGreen
data CarLight = CRed | CYellow | CGreen
data Switch = On | Off
data Signal = Present
```

Note that the channels may be empty, therefore we need no signal `Absent`. Figure 2 shows the structure of the system, which contains (non-critical) components to merge (ButMerge) and to split (AckSPlit, CLSPlit, PLSplit) the messages. The only critical component is the controller, which is developed more formally.

The critical requirements, which the controller has to fulfil in any case, are formalized using temporal logic (with the until-operator $\cup$ and the next-operator $\bigcirc$).

NO-CRASH: $\Box\ \neg$(CL=CGreen $\wedge$ PL=PGreen)
NO-BLOCK: $\Box\ \Diamond$ CL=CGreen $\wedge\ \Box$ (But=Present $\Rightarrow\ \Diamond$ PL=PGreen)
SEPARATE: ((CL=CGreen $\cup$ (CL=CYellow $\wedge\ \bigcirc$CL=CRed)) $\cup$ CL=CRed) $\cup$ CL=CYellow

Of course, there are many more requirements that are less safety critical such as that the pedestrian light only gets green if the button is present.

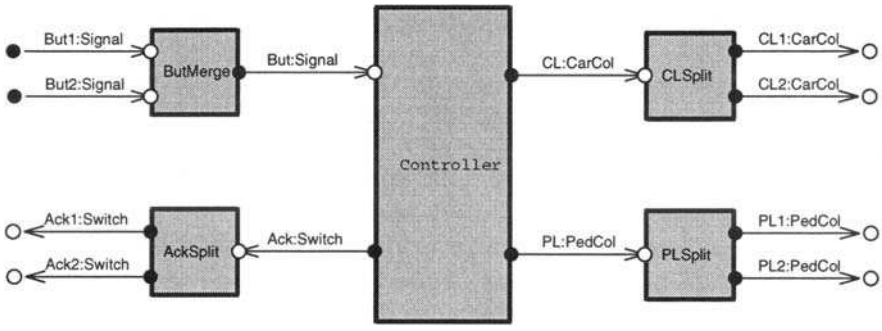


Fig. 2. Structure of Traffic Light System

Note that in **NO-BLOCK** $\Diamond$ PL=PGreen does not always hold. If no pedestrian arrives, the light will never become green. The specification of temporal logic formulas, especially using the until-operator U is not an easy task. Therefore more readable notations are useful. Interaction description diagrams are an intuitive way to specify some temporal formulas. However, they are not as expressive as the temporal logic (see below).

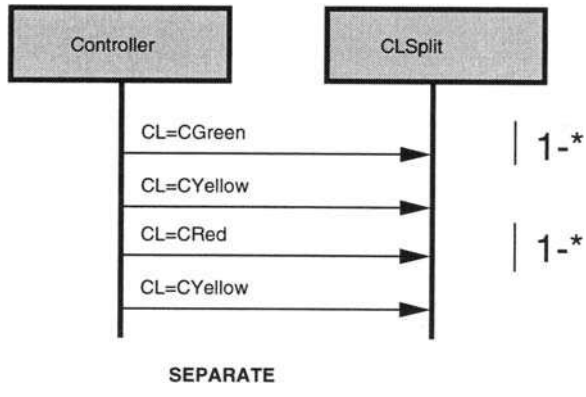


Fig. 3. Visualization of Separation Property

The critical aspect **SEPARATE** has a representation as an interaction diagram. In Figure 3 the (strong) until-operator U is visualized by the (1 - *) modifier in the interaction diagram. Note that we omitted the outputs of the compo-

nent CLSplit in the diagram, because CLSplit is not a critical component, and we therefore do not formally specify its behaviour[1]. Using interaction description diagrams like EETs (extended event traces [BHKS97]) or MSCs [IT93b] in this way is only possible if there exists a formal semantics (see [BHKS97,Bro98,Hau97] for semantics of EETs and MSCs).

The properties **NO-CRASH**, and **NO-BLOCK** cannot be represented so easily by EETs or MSCs. It is a current research topic to extend expressiveness of graphical description techniques towards more complex properties. One straightforward extension of EETs is the visualization of simultaneous actions, as developed within the project Quest. It uses time ticks to group messages together that occur within the same time interval. In our example we specify the crash-situation in Figure 4. Note that this property must not occur in our system.

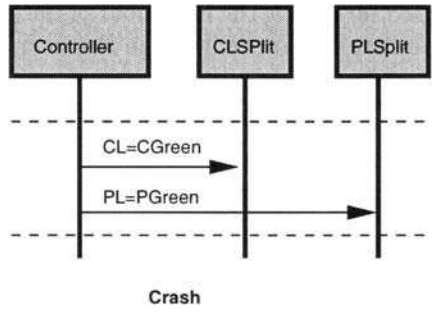


Fig. 4. Visualization of Crash Property

Deciding during the specification of requirements what a critical component is, the user can be supported by searching all components of the system that produce messages mentioned in the specification of critical properties. In our example this are the components ButMerge and Controller. We decided not to regard ButMerge as critical, since its output is used only as a premise within the property **NO-BLOCK**.

In the case of a more abstract security model it might be necessary to map the general properties to the concrete components. In our example the property that no car hits a pedestrian has been mapped informally to the controller property **NO-CRASH**. Ensuring adequacy of this mapping would require to build a model of the application domain, including assumptions to the environment, like every car stops at a red light, or pedestrians leave the crossing no later than 5 seconds after they have entered it. In our small example we do not model the environment.

[1] It is also possible to omit the component CLSplit from the interaction diagram completely, but we want to show the interaction within interaction diagrams.

5.2 Design Specification

The second class of views of developed systems occur in the design phase. They capture the structure of the system (together with all channels and data types) and a concrete description of the behaviour of the components. For the critical components we require that there is a formal design specification in order to verify that the requirements are fulfilled (see Section 6.2). To ensure quality of non-critical parts a coarse formal specification of the design of the behaviour can help to develop test cases (see Section 6.4).

We use system structure diagrams (SSDs) to specify the design structure (as in the requirements specification) of the system, and state transition diagrams (STDs) to model the behaviour of components, since they have the same mathematical basis and therefore semantically fit the techniques of the requirements specification.

In our example we refine the structure of the controller into a core controller and a timer, which is used for timing the behaviour of the system. This structure, and the communication channels are depicted in Figure 5. The timer component

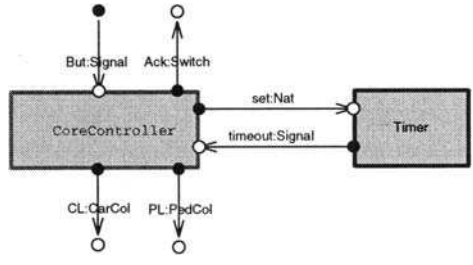


Fig. 5. Structure of Controller

uses the type of natural numbers, which is specified with the some functions working on them in the following data type specification:

```
data Nat = Zero | Succ(Nat);
fun pred(Succ(n)) = n;
```

The behaviour of the timer is specified by the STD in Figure 6, using tuples of preconditions, input, and output patterns and actions as transitions. The specification of the behaviour of the core controller is shown in the STD of Figure 7. It shows the two main states of the system: The state Wait, where pedestrians have to wait, and the state Walk where they may start walking. The transitions between the states show how the pedestrians lights are switched. The behaviour in Figure 7 is not completely designed, because it does not specify how the controller reacts on input, and how it switches the cars light. These details are modelled

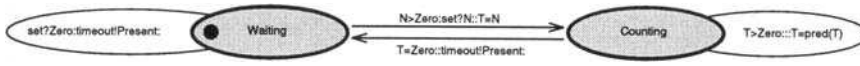


Fig. 6. Behaviour of Timer

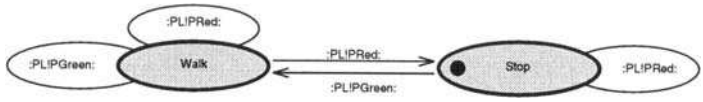


Fig. 7. Behaviour of CoreController

within the refined (sub-)STDs for the states Walk (Figure 8) and Wait (Figure 9). The introduced hierarchy is only syntactic, i.e the transitions (segments) are connected, such that the control in a hierarchic automaton is exactly in one atomic state. All transitions (segments) between different levels of abstractions are executed within one step. For example the transition from the atomic state Yellow to the atomic state Red leaves the hierarchic state Wait and enters the hierarchic state Walk and generates the output: CL!CRed,Ack!Off,PL!PGreen,set!ten

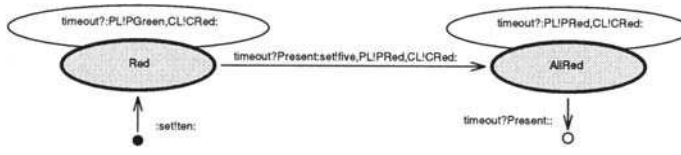


Fig. 8. Behaviour of Walk

Hierarchy is a very useful feature in formal notations. It allows us to specify the critical components at the right level of abstraction. Furthermore hierarchic formalisms support the development of larger systems. This is extremely important if graphical notations are used, since complex diagrams are hard to understand.

5.3 Implementation Specification

The third class of views on systems occur in the implementation specification, and describe the program code. They cannot be clearly separated from the views during the design phase. The main difference is that a design specification does

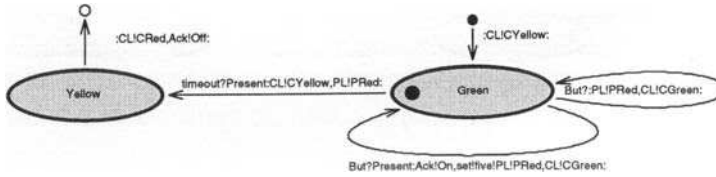


Fig. 9. Behaviour of Stop

not need to be deterministic, but an implementation and its specification should be deterministic; otherwise the test results may not be repeatable.

The program code can be seen as the most detailed design specification. There are at least two different possibilities to formally specify the implementation: one is to use a programming language with formal semantics, the other way is to use the same formal description techniques as in the design specifications and to generate code out of them. Many description techniques support code generation (Statecharts [Har87], SDL [IT93a], and STDs [HS97]).

Since a common semantics for all views is required, we recommend again to use STDs for the specification of the code. In our example the detailed design specification of the core controller (Figures 7, 8, and 9) is deterministic, and we can generate code out of it. The component Timer was classified to be non-critical for the following reasons:

- It has an infinite state space (it contains a variable `T:Nat`, to store the remaining time), and therefore it cannot be model checked completely. Treating the timer as critical would require interactive theorem proving (see Section 6), which in our case is not adequate.
- Treating it as non-critical allows us to implement it manually, for example using a realtime-clock that ensures that the distance between two ticks is no less that 100 ms[2].

We restrict our implementation specifications to deterministic programs. For non-critical parts of the system any (deterministic) programming language can be chosen, the only important point is that the interfaces fit to the formal interface specification.

5.4 Test Specification

Our last class of views in the development process describe test cases. Testing components is only necessary if they are developed informally. Specifying test cases is useful for documentation purpose, and a specified test case can be reused if the implementation has changed. In Section 6.4, we show how test cases can be generated from specifications.

[2] The upper bound depends on the cycle time of our system and cannot be ensured formally within our time-free model.

As a representation of a test case we choose interaction diagrams (with time ticks), as they were presented in Section 5.1, but we do not require the full power of them, since for testing one component or one system an interaction diagram with one axis suffices. Furthermore we require to have concrete input values (no variables) on the input messages, to test the component.

5.5 Description Techniques

The description techniques used during the development process are summarized in the following table:

name	purpose	phases	hierarchy	semantics
DTD	data types	all phases	inclusion of DTDs	algebras
SSD	structure	all phases	SSDs for components	streams & functions
STD	behaviour	design, implement.	STDs for substates	IO-automata
EET	Interaction	requirement, test	EETs in boxes	predicates

Note that we did not use the inclusion of data types, nor the grouping of EETs into boxes in our example. However, since hierarchic description techniques support different levels of abstraction it is important that all description techniques are hierarchical.

AutoFocus [HMR+98] is a tool that supports the specification, with all these description techniques and has features like consistency checks and simulation. The common semantical basis of the description techniques is described in [BDD+93,Bro95].

6 Development

In the previous section we presented graphical support for the specification of the different views, which occur at different stages within the development process. Furthermore we integrated formal methods into conventional software engineering in a way that only the critical parts have to be formally modelled. The used description techniques have a formal semantics. In this section we will exploit the fact that this semantics is uniform, i.e. all description techniques have a denotation within the same semantic model. This allows us to define relations between different description techniques. The fact that these relations can be formally checked provides us with a powerful tool to develop correct software.

6.1 Checking Requirements against Design

In this section we present methods that allow us to formally verify whether a design specification meets its requirements specification. We discuss different specifications and present different techniques for tool supported correctness proofs.

The most simple case is a deterministic design and simple temporal formulas, which can be represented by EETs. This situation corresponds to a formal test. To "verify" this it suffices to feed the input values according to the specification of the test case (EET) into a simulation of the component and to compare the outputs. If the outputs fit together the test is successful. One example is the requirement **SEPARATE**, specified in the EET of Figure 3. Simulating the design specification "proves" that this requirement is fulfilled by the specification.

A slightly more difficult case is that of non-deterministic automata and non-deterministic (but finite) choices in the interaction description diagrams. It requires backtracking if the output of the component does not fit to its requirements. Backtracking checks all alternatives. If they are finite, backtracking can decide whether the requirements hold. Since the programming language Prolog supports backtracking, it seems appropriate for this form of verification[3]. If the alternatives within a program are not finite, for example due to an input value with infinite range Prolog might find a solution. But it might also not terminate, if no solution exists.

If the requirements are specified with temporal logic, especially using a negation (like in **NO-CRASH**), simulation has to inspect all cases to ensure that the requirement is valid. In this case model checkers are more efficient than Prolog is. This case is standard in model checking using temporal logic and a finite state model of the behaviour. In our example the properties **NO-CRASH**, and **NO-BLOCK** could be verified with the model checker.

The most difficult case for the verification of requirements specified with temporal logic are infinite (or very large) models. In this case interactive theorem proving is appropriate. Using abstraction techniques allows us to reduce the model, and to produce simple proof obligations. However it is not easy to find the right abstractions that maintain the correctness of the model with respect to the desired property.

Model checking is today the only industrial accepted proof technology. However, without abstraction the application is restricted to the class of small and finite systems. Fortunately many embedded systems belong to this class.

6.2 Refinement

Refinement defines the relation between the requirements specification and the design as well as between the design and the implementation model. The refinement relation [Bro95] is transitive, such that a stepwise development is possible, and monotonic for the composition operators to support a modular development. In this section we present some development steps and the corresponding techniques to ensure correctness.

The easiest refinements are straightforward structural refinements, since they do not require to prove correctness. One example is the refinement of the component Controller in Figure 2 by the structure given in Figure 5. The definition of a behaviour for the components CoreController and Timer is also a refinement

[3] Within project Quest we are evaluating Prolog for this purpose.

step without proof obligation. In the case the refined structure (in our case Controller) has a behaviour associated, it has to be proved that the refined structure or behaviour is indeed a refinement (*behavioural refinement*). In our example the refinement of the states (Figure 7) by the automata describing the substates (Figure 8, and 9) are behavioural refinements.

The proof obligations of behavioural refinement can be proved by model checking (in finite and small cases) or by interactive, deduction based theorem proving (in the general case). Both proof techniques ensure that the behaviour of the refined component is correct with respect to the behaviour specification of the abstract component. Other typical examples are the elimination of underspecification, or the implementation of one data structure by a more efficient one.

Special cases for behavioural refinements are abstractions between two models that occur during the requirements verification, and the refinement of interfaces, which also bases on abstraction techniques [Slo97]. With these development steps every step of the software engineering process can be formally verified, however, since formal refinement is a time consuming activity the effort can be reduced by refining only critical components formally.

In the development process (see Figure 1) there are informal steps from the abstract views towards more concrete views. The formalization by the semantic model and its representation in logics allows us to prove the correctness of such steps. In some situations it might be helpful to do some new refinement steps within the formal model, and to translate the result back to graphical descriptions in order to continue with a semi-formal development. In the project Quest (see Section 7) we provide partial retranslations from formal into graphical description techniques.

6.3 Code Generation

As mentioned in Section 5.3 there exist a generation of code from "executable specifications". One of the crucial challenges in the application of formal methods is to formally prove the correctness of code generators [BBF+92].

6.4 Generation of Tests

In Section 5.4 the general representations for test-cases are described. In this section we focus on the generation of test cases from specifications.

As mentioned in the previous section refinement is quite time consuming and expensive, even compared with producing a formal specification of a system. It might be useful to specify the non-critical components formally, even if no refinement is intended. The reason for this is that a formal specification, even if it is only an abstract interface specification can serve as a good basis for test case generation.

The process of testing is done in the following steps:

1. selection of test cases and input values,

2. determination of output values for all (combinations of) input values,
3. representation of the test (documentation and reuse), and
4. testing the implementation.

There are many methods that allow us to derive test cases from the specification of the components. One popular method is the transition tour which suggests to find a test suite that covers all transitions of the automata describing the behaviour. Problems of this method, developed for automata without input and local variables, are to find the right input values and to ensure that the suggested transitions are possible. We do not go into the details of the test generation here we just refer to the work of the project Quest, especially [Sad98], and show the result of the test cases generation for the timer.

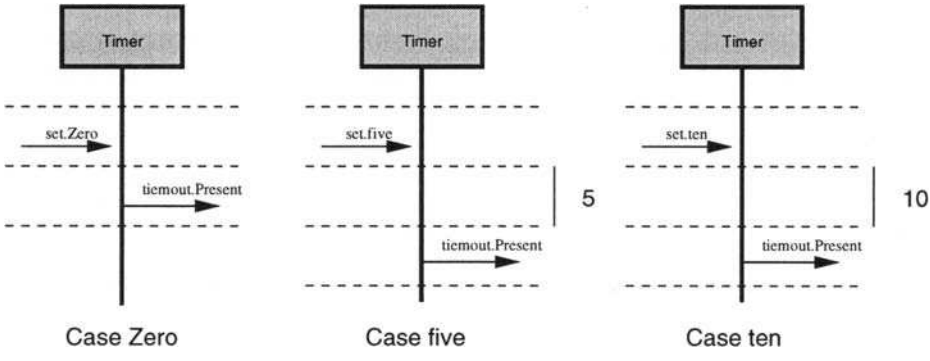


Fig. 10. Test cases for Timer

The input values were selected as a classification of natural numbers. Note that the time-ticks are essential in the diagram, since there is no (observable) signal to express time progress. If the implementation of the timer is deterministic and fulfils our test cases, the correctness of the critical component is ensured.

7 Conclusion and Future Work

We presented an extension of a conventional process model, with integrated formal methods, as far as it is desired, into the software engineering process. The technical foundation for this process model is the uniform semantic basis for the description techniques used within the development process. We demonstrated this technique by an example which we developed with user-oriented description techniques for structure, behaviour, and interaction, based on a model for data types. Our technique of a common semantic basis may also be applied to other description techniques to yield analogous results.

Note that the expressive power of the formal model influences the provable properties. If, for example, real-time properties have to be proved, an adequate

model is required, however, a complex model is often more difficult to handle. Finding expressive and practical models is a challenge in research. We decided to use a simple but extendable model for our description techniques that allows us to develop embedded systems and that can be easily extended to more complex applications like real-time, hybrid [MS96], dynamic, and object-oriented [BGH+98] systems.

The biggest part of the effort in applying formal methods is the work in proving the correctness of refinement steps. Since the costs for learning and using formally founded description techniques is relatively small we recommend to use them in many parts of the development for the following reasons:

- Formally founded description techniques are a clear and precise way to model different views of the system.
- It is easier to increase the applications of formal methods within a development project, for example by deciding that a former non-critical component becomes critical.
- Specification based test methods can be applied to generate test cases systematically.

The future work can be divided into two directions. One is the formalization of additional description techniques [BCMR97,Hin98]. However, since many graphical description techniques, especially from UML, have many features, the required semantic model becomes complex [BGH+98] and this will make refinement steps difficult. It is therefore crucial to increase the expressive power of theorem provers and model checkers to handle complex models.

Another direction of future work is to provide tools for the formalization of description techniques with a simple and common semantic basis. In our Munich research group this direction is pursued within the project Quest. The goal of the project Quest is to provide tools for the co-development of embedded systems. To achieve this goal we concentrate on the description techniques presented in this paper, and we connect AUTOFOCUS, the modeling tool with model checkers and the theorem proving environment VSE II [RSW97]. For the development of non-critical parts of the systems we provide a test environment with specification based test generation techniques.

Acknowledgment: For many helpful comments on previous versions of this paper we thank Katharina Spies, Bernhard Schätz, and Peter Braun.

References

[BBF+92] Bettina Buth, Karl-Heinz Buth, Martin Fräzle, Burghard von Karger, Yassine Lakhneche, Hans Langmaack, and Markus Müller-Olm. Provably Correct Compiler Development and Implementation. In Uwe Kastens and Peter Pfahler, editors, *Compiler Construction, 4th International Conference on Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*, pages 141–155, Paderborn, Germany, 5–7 October 1992. Springer.

[BCMR97] Manfred Broy, Derek Coleman, Tom S. E. Maibaum, and Bernhard Rumpe, editors. *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques.* Technische Universitaet Muenchen, TUM-I9803, April 1997.

[BD95] Adolf-Peter Bröhl and Wolfgang Dröschel. *Das V-Modell.* Oldenbourg, 1995.

[BDD+93] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The Design of Distributed Systems: An Introduction to Focus—Revised Version. Technical Report TUM-I9202-2, Technische Universität München, 1993.

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In Martin Schader and Axel Korthaus, editors, *The Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, 1998.

[BHKS97] Manfred Broy, Christoph Hofmann, Ingolf Krüger, and Monika Schmidt. Using extended event traces to describe communication in software architectures. In *Asia-Pacific Software Engineering Conference and International Computer Science Conference, Hong Kong.* IEEE Computer Society, 1997.

[BJ95] Manfred Broy and Stefan Jähnichen, editors. *KORSO: Methods, Languages and Tools for the Construction of Correct Software*, volume 1009 of *Lecture Notes in Computer Science*, New York, N.Y., 1995. Springer-Verlag.

[Bro95] Manfred Broy. Mathematical Models as a Basis of Software Engineering. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 292–306. Springer-Verlag, 1995.

[Bro96] Manfred Broy. Formal Description Techniques - How Formal and Descriptive are they. In R. Gotzhein and J. Bredereke, editors, *FORTE IX, 95-112.* Chapman & Hall, 1996.

[Bro98] Manfred Broy. On the Meaning of Message Sequence Charts. In Lahav, Wolisz, Fischer, and Holz, editors, *1st Workshop on SDL and MSC (SAM98)*, pages 13–32, 1998.

[GSB98] R. Grosu, G. Stefanescu, and M. Broy. Visual Formalisms Revisited. In L. Lavagno and W. Reisig, editors, *CSD '98, International Conference on Application of Concurrency to System Design, Aizu-Wakamatsu City, Fukushima.* IEEE Computer Society Press, 1998.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231 – 274, 1987.

[Hau97] Markus Haubner. Transformation von MSCs in temporallogische Formeln, 1997. Diplomarbeit.

[Hin98] Ursula Hinkel. *Formale, semantische Fundierung und eine darauf abgestützte Verifikationsmethode für SDL.* PhD thesis, Technische Universität München, 1998.

[HMR+98] Franz Huber, Sascha Molterer, Andreas Rausch, Bernhard Schätz, Marc Sihling, and Oscar Slotosch. Tool supported Specification and Simulation of Distributed Systems. In Bernd Krämer, Naoshi Uchihira, Peter Croll, and Stefano Russo, editors, *Proceedings International Symposium on Software Engineering for Parallel and Distributed Systems, pp. 155-164, ISBN 0-8186-8467-4*, pages 155–164. IEEE Computer Society, Los Alamitos, California, 1998.

[Hoa96] C. A. R. Hoare. The Role of Formal Techniques: Past, Current and Future or How Did Software Get so Reliable without Proof? In *18th International Conference on Software Engineering*, pages 233–235, Berlin - Heidelberg - New York, March 1996. Springer.

[HS97] Franz Huber and Bernhard Schätz. Rapid Prototyping with AutoFocus. In A. Wolisz, I. Schieferdecker, and A. Rennoch, editors, *Formale Beschreibungstechniken für verteilte Systeme, GI/ITG Fachgespräch 1997, pp. 343-352.* GMD Verlag (St. Augustin), 1997.

[IT93a] ITU-T. *Recommendation Z.100, Specification and Description Language (SDL).* ITU, 1993.

[IT93b] ITU-T. *Recommendation Z.120, Message Sequence Chart (MSC).* ITU, 1993.

[MS96] Olaf Müller and Peter Scholz. Specification of Real-Time and Hybrid Systems in FOCUS. Technical Report TUM-I9627, Technische Univerität München, 1996.

[Par95] D.L. Parnas. Using Mathematical Models in the Inspection of Critical Software. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Applications of Formal Methods*, International Series in Computer Science, chapter 2, pages 17–31. Prentice Hall, 1995.

[RSW97] Georg Rock, Werner Stephan, and Andreas Wolpers. Tool Support for the Compositional Development of Distributed Systems. In *Tagungsband 7. GI/ITG-Fachgespräch Formale Beschreibungstechniken für verteilte Systeme*, number 315 in GMD Studien. GMD, 1997.

[Sad98] S. Sadeghipour. *Testing Cyclic Software Components of Reactive Systems on the Basis of Formal Specifications.* PhD thesis, Technische Universitt Berlin, Fachbereich Informatik, 1998.

[Slo97] Oscar Slotosch. *Refinements in HOLCF: Implementation of Interactive Systems.* PhD thesis, Technische Universität München, 1997.

Formal Program Development in Geometric Modeling*

François Puitg and Jean-François Dufourd

Laboratoire des Sciences de l'Image, de l'Informatique et de la Télédétection
(LSIIT, UPRES-A 7005)
Pôle API, boulevard S. Brant F-67400 Illkirch Cedex, France
email: {puitg,dufourd}@dpt-info.u-strasbg.fr
www: http://dpt-info.u-strasbg.fr/lsiit

Abstract. An innovative attempt to integrate formal program development in geometric modeling is reported through the axiomatization of model of the combinatorial maps in the Calculus of Inductive Constructions. A hierarchical specification of ordered sorts is validated in the Coq prover by inductive proofs, and the automatic extraction of a prototype. Classical difficulties – like cohabitation of hierarchized objects, smooth handling of subtyping, and completion of partial relations – are addressed both from theorem proving and prototyping viewpoint.

1 Introduction

The computer graphics research team in Strasbourg aims at to get a general framework allowing construction and handling of sound geometric objects on computers, via formal methods. Algebraic specifications [1] have been intensively used, especially to prototype from scratch *Topofil* – an interactive 3D modeler [2] – and to formalize the foundations of geometric modeling [3], thanks to the OBJ3 language [4]. But impediments have arisen in the use of these algebraic methods, in particular with regard to ordered sorts and preconditions, and poor proving support. We thus attempt to extend these works to the dialectics of constructive proofs and to prototyping, making lengthful use of induction [5] and program extraction. This is a real innovative approach in geometric modeling. Though many models have been proposed, few have been really investigated in depth – except in the survey [6], but in the usual unformal abstract way of mathematicians – or validated and tested within the same logical framework.

At present, we want to axiomatize the particular model of the combinatorial maps [7], set up a necessary and sufficient condition of planarity to investigate Euler's formula and the theorem of the genus, work out a discrete Jordan's theorem [8], and extract a prototype. The theorem of the genus was proved by Jacques, but from an unsafe generalization of a Serret's lemma [9]. This was corrected by Cori [10], despite the fact that several arguments are intuitive and

* This work is supported by the GDR-PRC of *Programmation*, and the GDR-PRC of *Algorithmique, Modles et Infographie* (MENRT, CNRS, France).

should be clarified. One can find another proof by Tutte [8], but from such a general point of view that it seems impossible to get a clear idea of the variation of the genus. None of these works deals with the integration of validation-through-testing into their approach. As for Jordan's theorem, it is one of the oldest archetypal issues of geometric modeling. Exposing it in a new up-to-date fashion with adequate theories, solving it on a prover and testing it via a prototype will result in a great insight and more generally will shed light on other domain of imagery or on software development [2].

This paper thus reports an axiomatization of combinatorial planar maps in boundary representation – presented in Sect. 2 – within the Calculus of Inductive Constructions, exposed in Sect. 3. Starting with free maps, we proceed with quasi-maps, then with combinatorial maps in the hierarchical specification of Sect. 4. Section 5 focuses on how program extraction can be smoothly integrated in the axiomatics to perform prototyping. The combinatorial characteristics are tackled in Section 6. Section 7 concludes and opens up prospects.

2 Boundary Representation

The aim of geometric modeling is to describe geometric objects and operations. Historically, the Boundary Representation (B-rep) was investigated in order to represent solids that can be manufactured on a machine tool, i.e. solids whose boundary makes an orientable closed surface. The topology of the boundary is defined by a subdivision into cells – vertices, edges, faces – and incidence relations between them. An embedding model then associates for example vertices with points, edges with curves and faces with surfaces.

Among the numerous models formalizing the B-rep, we have chosen the combinatorial maps. They represent the topology of polyhedra homeomorphic to the torus with g holes. If $g = 0$, we deal with planar maps. Usually, a projection on the plane is used for drawings (except Fig. 2), with intersections if $g \neq 0$.

2.1 Combinatorial Maps

In the following, all the elements of the definitions belong to a finite set $\mathcal{D}$.

Recall a *relation* $\mathcal{R}$ is a collection of ordered pairs (x, y) such that x is associated with y by $\mathcal{R}$, denoted $(\mathcal{R}\ x\ y)$. Then, y is a *successor* of x w.r.t $\mathcal{R}$, or $\mathcal{R}$-successor of x, and x a *predecessor*, or $\mathcal{R}$-predecessor, of y. $\mathcal{R}$ is *total* if for all x there exists y such that $(\mathcal{R}\ x\ y)$. Otherwise, it is *partial*. We say that $\mathcal{R}$ is *involutive* if for all x, y, z such that $(\mathcal{R}\ x\ y)$ and $(\mathcal{R}\ y\ z)$, $z = x$, *injective* if for all x, x', y, such that $(\mathcal{R}\ x\ y)$ and $(\mathcal{R}\ x'\ y)$, $x' = x$, *surjective* if for all y, there exists x such that $(\mathcal{R}\ x\ y)$. It is a (partial) *function* if for all x, y, y', such that $(\mathcal{R}\ x\ y)$ and $(\mathcal{R}\ x\ y')$, $y' = y$. An *application* is a total function. An *involution* (resp. injection, surjection) is an involutive (resp. injective, surjective) application. An injective and surjective bijection from $\mathcal{D}$ to $\mathcal{D}$ is called a *permutation*.

A *combinatorial map* is defined as a triple $(\mathcal{D}, \alpha_0, \alpha_1)$ where $\mathcal{D}$ is a finite set, α_0 an involution and α_1 a permutation, in $\mathcal{D}$ [7]. An element of $\mathcal{D}$ is called

a *dart*. For $k = 0, 1$, k-successor (resp. k-predecessor) stands for α_k-successor (resp. α_k-predecessor). Triple (k, x, y) such that $(\alpha_k\ x\ y)$ is called a *k-sewing*. Its darts are said *k-sewn*. When the α_k are just functions, we speak of *quasi-map*.

Figure 1 shows the graph of a quasi-map embedded in the plane. Darts are represented as half-segments numbered by integers, α_0 is symbolized with thin perpendicular dashes between the two corresponding darts and α_1 with black spots. Darts are ordered counterclockwise around dashes and spots, w.r.t. to their succession in α_0 and α_1 respectively. The thin perpendicular dashes can halve and the black spots become arcs of circle when the α_k are partial.

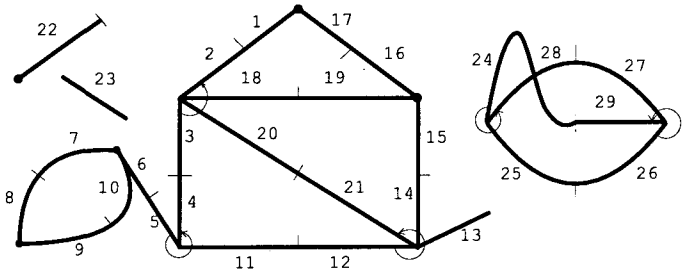


Fig. 1. A quasi-map. $\mathcal{D} = \{1, 2, \ldots, 29\}$, and for example $(\alpha_0\ 22\ 22)$, $(\alpha_0\ 1\ 2)$, 1 has no 0-predecessor, $(\alpha_1\ 24\ 25)$, $(\alpha_1\ 1\ 17)$, 2 has no 1-successor, 23 is not sewn at all

2.2 Planar Maps

We just give intuitive definitions, formal ones can be found in [11]. Components of maps are classically defined as equivalence classes, here w.r.t. α_0 and α_1. In Fig. 1, the sets $\{22\}$ and $\{24, 25, 26, 27, 28, 29\}$ are two of the four components.

The orbits of a function $\mathcal{R}$ are the component of the graph of $\mathcal{R}$. If $\mathcal{R}$ is partial, they are open or *incomplete*, otherwise closed or *complete*. Then, since they are cyclic, any circular permutation denotes the same orbit. The set of the orbits of $\mathcal{R}$ is denoted $<\mathcal{R}>$. For instance, on the previous figure: $[22] \text{ and } [1, 2] \in <\alpha_0>$; $[22] \text{ and } [24, 25, 28] \in <\alpha_1>$; $[22] \text{ and } [19, 20, 14, 19] \in <\alpha_1^{-1} \circ \alpha_0>$.

Let $\mathrm{m} = (\mathcal{D}, \alpha_0, \alpha_1)$ be a combinatorial map. Interesting orbits of m concern respectively α_0, α_1 and $\alpha_1^{-1} \circ \alpha_0$. They contain cycles, for α_0, α_1 and $\alpha_1^{-1} \circ \alpha_0$ are bijections. An element of $<\alpha_1>$ (resp. $<\alpha_0>$, $<\alpha_1^{-1} \circ \alpha_0>$) is a *vertex* (resp. an *edge*, an *oriented face*) of m. Were the quasi-map of Fig. 1 be completed, it would have 16 edges, 11 vertices and 8 faces, including the *external face* $[2, 17, 15, 13, 12, 5, 10, 8, 6, 4, 2]$ of the biggest component.

Let nc (resp. nd, nv, ne, nf) be the number of components (resp. darts, vertices, edges, faces) of a map m. Euler's characteristic χ and the *genus* g are defined by $\chi = \mathrm{nv} - (\mathrm{nd} - \mathrm{ne}) + \mathrm{nf}$ and $g = \mathrm{nc} - \chi/2$. Intuitively, Euler's characteristic may be interpreted as the minimum number of poles necessary to mesh the corresponding surface: 2 for a sphere, 0 for a torus. The genus of a map gives the number of holes of the polyhedra structure whose boundary is modeled by that map: 0 for a sphere, 1 for a torus (Fig. 2).

The theorem of the genus [8–10] – $g \in \mathbb{N}$ – makes this interpretation plausible. Now, a *planar map* is a map whith $g = 0$, i.e. which satisfies Euler's formula:

$2nc + nd = nv + ne + nf$. The completed of the quasi-map in Fig. 1 is not planar. Indeed, $g = 4 - (11 - (29 - 16) + 8)/2 = 1$. The non-planarity is introduced by the component $\{24, 25, 26, 27, 28, 29\}$. The other components are planar.

Fig. 2. Maps modeling the topology of a subdivision of a sphere (resp. a torus) containing 1 vertex, 1 edge and 1 face (resp. 1 vertex, 2 edges and 1 face). Vertices (resp. edges, faces) are embedded on bold crosses (resp. bold curves, meshed surfaces)

3 Using the Calculus of Inductive Constructions

Our axiomatizations are expressed in the *Calculus of Inductive Constructions* (CIC) [12], a powerful higher-order logical framework. It is able to recover our algebraic specification background into a single formalism, in which our past difficulties with ordered sorts and preconditions may be solved, and our new ambitions with regard to proofs and program extraction supported.

Our proofs are constructed in *Coq* [1] [13], a theorem proving assistant built on CIC. A high-level notation, the *Gallina* language – sufficiently expressive to get a simple though unambiguous coding, close to the usual mathematical notation universally understood – allows the declaration of axioms, the definition of types and objects and a hierarchical development of our theories.

An interactive dialogue with the prover is achieved by the application of predefined or user-programmable *tactics* – the implementation of logical reasoning steps – in a top-down manner, subgoaling recursively. Proofs developments in Coq are informative, so functional programs can be associated with our proofs to perform quick prototyping, during the *program extraction* process [14].

In the following, Gallina scripts are typewritten, keywords are underlined, types are italicized, new identifiers in definitions are boldfaced, and names of theorems are CAPITALIZED. Proof scripts, definitely too wordy, are omitted.

4 A hierarchy of Maps

Proving often requires objects more general than the ones in play. Thus, we use a hierarchy of nested types, with general objects and atomic constructors at the top and specialized objects and complex constructors at the bottom. The first level is a classical specification of what we call *free maps*. They are the most general objects, without any constraint or precondition. The second one is inhabited by *quasi-maps*, obtained by restriction of the free maps with a predicate of

[1] Coq is a research project common to INRIA, ENS Lyon and CNRS in France, and part of the European Esprit working group 21900 - TYPES.

well-formedness, in order to meet geometric modeling requirements. The third one houses the *combinatorial maps*, obtained by completing chosen selectors. Usually, when we informally think about some concept, we do it in terms of quasi-map whose orbits are supposed completed or not. This only provides a thinking support, Gallina definitions always take place on free maps.

Let us give first some basic notions: dimension of sewing, dart and sewing.

4.1 Sewings and Darts

For defining the sewing dimensions 0 and 1, we add a new inductive data type dim, as the least Set containing the constructors zero and one. Set is the built-in type of concrete sets such as booleans, natural numbers, lists, ...:

DEFINITION 1 (DIMENSIONS). A sewing dimension, dim , is either zero or one:

```
Inductive dim : Set
  := zero : dim
   | one : dim
```

Notation t:T reads "t is of type T". Because some proofs bring in sewing dimensions variables, we have to be able to distinguish them. This – as well as the structural induction principles over dim objects, automatically provided by Coq – is heavily used to prove statements by case analysis in proof mode, using the following reasonning scheme: "Suppose k and k' are equal, ... Now if they are different, ...".

THEOREM (DECIDABILITY OF DIMENSIONS EQUALITY). For all dimensions k, k', either the property k=k' holds, or it does not, i.e. the property "not k=k'" holds:

```
Theorem EQ_DIM_DEC : ∀k,k':dim  {k=k'} + {~k=k'}
```

Instead of the classical propositional connective or denoted ∨, we use the intuitionistic disjunction sumbool denoted "{}+{}" [13, page 101], pointing our future intention to perform program extraction out.

DEFINITION 2. Darts are just declared as the elements of a parameter dart, the so-called set of darts. They are supposed distinguishable:

```
Parameter dart : Set. Axiom EQ_DART_DEC : ∀x,y:dart {x=y}+{~x=y}
```

A sewing dimension and two darts make a sewing. To handle sewings like any other objects, we encapsulate these three components, thus getting a homogeneous and modular approach. The corresponding type, constructed by c, is denoted sw. For example (c zero x y):sw . As for dimensions , we need to distinguish sewings, thanks to the theorem EQ_SW_DEC, analogous to EQ_DIM_DEC.

4.2 Free Maps

In the construction of a free map, we want to be able to insert a dart or to sew two darts in order to build orbits step by step, without constraints or preconditions.

DEFINITION 3. The type of free maps is the smallest set denoted fmap containing the void free map (v), closed by insertion of darts and sewings (i, l):

```
Inductive fmap : Set
  := v : fmap
   | i : dart → fmap → fmap
   | l : sw → fmap → fmap
```

In free map m , (i d m) inserts a dart d and (l s m) a sewing s. All subsequent inductive definitions involving a free map will follow this structure: to define a predicate – say P – on free maps, one has to say how P behaves on v and what is going on for P when inserting a dart or a sewing.

We want to know if a dart belongs to a free map, if a dart has successors or predecessors and which ones if it does. These operations may be thought in terms of functions, yielding True or False or some dart. First, though we could build them in Gallina with the Fixpoint definition, inductive definitions are easier to set up, read and handle in proofs because of the "smallest set" property. So we claim that predicates or relations are more adequate for axiomatics; fixpoints should be reserved when intending to prototype.

DEFINITION 4. A dart exists in a map if either it has just been inserted (exd_i1), or it was already in and inserting darts (exd_i2) or sewings (exd_l) does not change anything.

```
Inductive exd [x:dart] : fmap → Prop
  := exd_i1: ∀m:fmap  (exd x (i x m))
   | exd_i2: ∀m:fmap ∀d:dart  (exd x m) → (exd x (i d m))
   | exd_l : ∀m:fmap ∀s:sw  (exd x m) → (exd x (l s m))
```

Prop is the built-in type of propositions. Note the parameterization of x, between square brackets. Decidability is needed for the existence of a dart in a free map:

THEOREM (DECIDABILITY OF THE EXISTENCE OF DARTS).

```
Theorem EXD_DEC : ∀x:dart ∀m:fmap  {exd x m} + {~(exd x m)}
```

From now on, though systematically defined and used, we will not mention decidability results any more.

The existence of successors or predecessors of some dart at a given dimension is a positive information. We rather need a negative information stating that some dart does not have successors. The corresponding predicate, of type dim→dart→fmap→Prop, is denoted nosucc. It is inductively built, in the spirit of exd. For instance, if m is the quasi-map represented on Fig. 1, it can be proved that (nosucc zero 2 m) and ~(nosucc one 19 m). The non-existence of predecessors, denoted nopred, is defined in the same way.

To define successors, i.e. α_0 and α_1 in Sect. 63, we start from relations and obtain, in the same way as for Def. 4:

DEFINITION 5 (SUCCESSORS).

```
Inductive succ [k:dim;x:dart] : fmap → dart → Prop
  := succ_1 : ∀m:fmap ∀y:dart  (succ k x (1 (c k x y) m) y)
  | succ_i : ∀m:fmap ∀y,d:dart
                (succ k x m y) → (succ k x (i d m) y)
  | succ_1': ∀m:fmap ∀y:dart ∀s:sw
                (succ k x m y) → (succ k x (1 s m) y)
```

The fact that a successor can be exhibited and the non-existence of successors are mutually exclusive. In proof mode, this result is essential because it divides the reasonning into two subgoals, introducing a successor in the first one or the hypothesis that it does not exist in the second. At the prototyping level, it will extract into a function yielding a successor or the information that it cannot be found. So again, we use the intuitionistic disjunction, as well as the weak dependent sum [13, page 100] instead of the classical existential quantifier:

THEOREM (succ OR nosucc, DECIDABILITY).

```
Theorem SUCC_NOSUCC : ∀k:dim ∀x:dart ∀m:fmap
  {y:dart | (succ k x m y)} + {nosucc k x m}
```

The same kind of decidability holds between succ and nopred. These results, as well as a few others in [11], make the axiomatization of the free maps complete and consistent. At this stage, we have not yet the combinatorial maps, but a generalization of the direct 2-graphs.

4.3 Quasi-Maps

The abstract data type of free maps is too large from the geometric modeling point of view: e.g. two darts can be sewn though they have not been inserted or a dart can be inserted twice or can have multiple predecessors at the same dimension. To avoid these degenerated cases, preconditions should be imposed to get *well-formed* free maps, i.e. what we call *quasi-maps*. The corresponding predicate, of type fmap→Prop, is denoted wf. New properties like the functionality of succ on quasi-maps can be proved. The specification of quasi-maps follows, as a free map – called the *support* – and a proof of its well-formedness:

DEFINITION 6 (QUASI-MAPS).

```
Record qmap : Set
  := qmap_intro {support : fmap; wfsupport : (wf support)}
```

The Record macro stands for non recursive inductive definitions with only one constructor. Note that unlike in most programming languages, the type of a field in a record may depend on another field, as it is the case for the type of wfsupport that depends on support. The ability of Gallina to make proofs part of propositions and types dependant allows a smooth and elegant subtyping both in definitions and in proofs, and can be propagated at the extraction level (see Sect. 5). It is a way of getting rid of the numerous problems we have encountered

in algebraic specifications when defining subsorts by invariants. Quasi-maps may be seen as a subtype of free maps and used like any other type without making explicit the support and its well-formedness – as in m:qmap – or explicitly constructed with qmap_intro.

We are now able to prove several essential results. On quasi-maps, succ is an injective function. Besides, (succ zero) is involutive.

4.4 Combinatorial Maps

A combinatorial map is viewed as a quasi-map whose orbits are complete. A possible method, difficult to set up because it requires to rip sewings, is to complete quasi-maps adding sewings where they lack [11]. Another one, presented here, is to complete selectors in order to obtain total rather than partial relations.

The existence of darts is not affected by completion: if a dart exists in a quasi-map, so does it in the same quasi-map viewed as a combinatorial map, and the converse is true. On the other hand, if we imagine the orbits of the quasi-map in Fig. 1 completed, the one-successor of 2 is 3, the one-predecessor of 24 is 28, 23 is the successor and the predecessor of itself, the zero-successor of 2 is 1 and the zero-predecessor of 1 is 2.

Thus, we must specify the ends of orbits, i.e. heads – not described here – and *tails*. The corresponding inductive relation, of type dim→dart→fmap→dart→Prop, is denoted tl: e.g. (tl one 18 m 3), (tl zero 2 m 1), (tl one 23 m 23). Completing succ into the new relation csucc, of type dim→dart→fmap→dart→Prop, is now easy: if z' is a successor of z, then so it is for csucc, if z does not have any successors, then we pick up a tail of z to make one.

On quasi-maps, tl is an application. At the prototyping level, we get a function yielding tails. Idem for csucc, moreover injective and surjective. Besides, (csucc zero) is involutive.

A combinatorial map is a quasi-map observed with csucc instead of succ. This is the so-called *observational* approach. The question of knowing if we get all the combinatorial maps this way is irrelevant, because a formal answer would be outside our theory of maps, on the meta-logic level. All we can say is we could not be able to think of a combinatorial map not constructible with our hierarchy. We now own the living wage to prototype.

5 Prototyping by extraction

In Coq, extraction can only be performed on intuitionistic proofs with computational contents. For instance, a classical proof of A ∨ B does not contain information on either the provability of A or else the provability of B. On the other hand, it is possible to build either a proof of A or a proof of B from the proof of the intuitionistic disjunction {A} + {B}. Thus, informative statements specifying the existence of computational objects (Set) must be separated from the logical assertions specifying their properties (Prop). Roughly speaking, to

extract programs, Coq analyses, checks and removes the Prop part – considered as comments – only keeping the Set part.

We mainly extract decidability results and applications. The formers are important because they allow us to distinguish objects, to answer questions – for instance, is a free map well-formed – and they correspond to the "if then else" construction in the proofs-as-programs context. In the extraction mechanism, they give functions with a yes-or-no output. The latters implement effectively the paradigm of the development of certified programs, transforming, say any dart x into a dart y verifying a well-chosen property Q from the informative proof of $\forall x\ \exists y\ (Q\ y)$. The more accurate Q, the more convincing and well tuned the prototype. Both types may be freely mixed.

The methodology is the following [14]: first, a constructive specification of the program to be extracted is given with a Theorem statement. Then, the user proposes a *realization* of this specification with the Realizer tactic. It is an algorithm expressed in a simple and syntactic-sugared language called Real, with constants, variables, λ-abstraction, application and recursion. The Program_all (PA) tactic makes Coq consider this algorithm as a sketch to prove the theorem, more or less without the user's help. In general, Coq finds proofs of informative goals alone, precisely because the realization can be considered as a skeleton of the proof. A few logical lemmas may stay at the end of this process. They must be interactively proved by the user. Finally, a functional program guaranteed correct w.r.t. its specification is automatically extracted.

5.1 Extraction of decidability

Dimensions We begin with the decidability of their equality. Given k,k':dim, we want to know whether k=k' or not. We need a kind of indexed boolean object working on *Prop*, but the usual propositional connective or (left) does not fit:

```
Inductive \/ [A,B:Prop] : Prop  Inductive {}+{} [A,B:Prop] : Set
  := or_introl : A → A ∨ B        := left  : A → {A}+{B}
  | or_intror : B → A ∨ B         | right : B → {A}+{B}
```

Indeed, since it is a proposition, it has no informative content and will be "forgotten" during extraction. On the contrary, the intuitionistic disjunction with constructive contents sumbool (right) – isomorphic to or, but yielding a Set – is extractible towards a concrete Caml type. Now, theorem EQ_DIM_DEC is a constructive specification of the program above. A realization of this specification is: if k=zero, then the left part of the specification must hold when k' is also zero, since zero=zero. Otherwise, the right part does, since ~zero=one. Vice-versa for k=one. This is expressed with the Real pattern-matching macro Cases:

```
Realizer λk,k':dim
  Cases k of
      zero ⇒ Cases k' of zero ⇒ left | one ⇒ right end
    | one  ⇒ Cases k' of zero ⇒ right | one ⇒ left end
  end
```

Note the syntax of the realizations: left and right correspond to the sumbool constructors. Their parameters A and B do not appear any more since they are

of type Prop. The type of the realization and the informative extraction of the goal, i.e. a Real-term of type dim→dim→sumbool must coincide. Therefore, bugs can be easily detected at this early proof stage, and corrected.

At the end of the work PA, EQ_DIM_DEC is proved. None logical statements are left to the user because this theorem, very simple, leads only to the informative subgoals we described above, easily solved by Coq. The automatically extracted Caml program has the following shape:

```
type sumbool = Left | Right; type dim = Zero | One

let eq_dim_dec k k' =
  match k with
    Zero → (match k' with Zero → Left | One → Right)
  | One  → (match k' with Zero → Right | One → Left)
```

As expected, the parameters of type Prop have disappeared in sumbool. The inductive definitions sumbool and dim are translated into enumerated types. Theorem EQ_DIM_DEC becomes the Caml functional value eq_dim_dec, whose structure strongly looks like the one of its Real realization. Note that names are maintained – except the first letter of constructors, automatically capitalized by Caml – and functions identifiers lower-cased.

Sewings The constructive specification of the program we want to extract is given by EQ_SW_DEC. A realization is: let (k,x,y) be s and (k',x',y') be s'. If k is equal to k', x to x' and y to y', then the left part of the specification must hold. Otherwise, the right part does, because ∼(c k x y)=(c k' x' y') whenever k, x or y differs from k', x', or y'. Thus, we have the Real realizer:

```
Realizer λs,s':sw
  let (k,x,y)=s in let (k',x',y')=s' in
    if (EQ_DIM_DEC k' k)
    then if (EQ_DART_DEC x' x)
         then if (EQ_DART_DEC y' y) then left else right
         else right
    else right
```

that Coq uses to easily derive a proof of EQ_SW_DEC. Extracting it would make Coq complain because neither the parameter dart nor the axiom EQ_DART_DEC have been realized. Let us instantiate darts on Caml integers: "ML Import int: Set. Link dart := int". Thus dart becomes a synonym for int. Then, we can give a computational contents to EQ_DART_DEC by attaching it to a Real-term:

```
Link eq_dart_dec :=
  λx,y:int if (eq_int x y) then left else right
```

Function eq_int stands for the Caml equality. Now, the automatic extraction in Caml of darts and sewings is possible, and gives:

```
type dart = int

let eq_dart_dec x y =
  match eq_int x y with true → Left | false → Right

type sw = C of dim × dart × dart
```

```
let eq_sw_dec s s' =
match s with C(k,x,y) → (match s' with C(k',x',y') →
  (match eq_dim_dec k' k with
    Left →
      (match eq_dart_dec x' x with
        Left →
        (match eq_dart_dec y' y with Left → Left|Right → Right)
        | Right → Right)
    | Right → Right))
```

Caml function eq_sw_dec is a bit complicated. We are not supposed to read it but just to use it in prototyping: its specification and realization are much easier to understand and more convincing since mechanically proved. So from now on, we do not expose Caml code anymore.

Existence of darts The decidability of the existence of darts in free maps is interesting because it calls on recursion. We want to prove and extract theorem EXD_DEC. Dart x does not exist in v, so we return right. It does exist in (i d mi) if d is equal to x, so left is returned. Otherwise, it exists in (i d mi) (resp. (l _ ml)) if it was the case before in mi (resp. ml) (recursive calls):

```
Realizer λx:dart
Fix exd_real {exd_real/1 : fmap → sumbool
:= λm  Cases m of
    v ⇒ right
   |(i d mi) ⇒ if (EQ_DART_DEC d x) then left else (exd_real mi)
   |(l _ ml) ⇒ (exd_real ml)
   end}
```

The expression Fix _{ _/1 : _ } defines a fixpoint exd_real with one free map argument, yielding sumbools. The whole type of the realization is thus dart→fmap→sumbool and matches the informative extraction of the goal. As usual, pattern-matching for destructuring m is performed with Cases. Unused arguments are omitted thanks to the underscore facility. The PA tactic leaves a few logical lemmas easily solved by the user [11].

The Caml extraction of EXD_DEC is yet a useful piece of prototype. Suppose m is a Caml value representing the free map of Fig. 1, then we may ask the interactive Caml toplevel "# exd 3 m;;". Its answer is "– : bool = true". Note the constructors Left and Right of sumbool do not appear. We have mapped them on true and false. The name exd is also a mapping of EXD_DEC. Indeed, exd better denotes what must be visible at the Caml level: the concept of the existence of darts. It refers to what really matters here: Caml exd is an executable prototype of the Coq relation exd.

5.2 Extraction of functions

Successors They are constructively specified in theorem EX_SUCC_2 by a sumor, which looks like a sumbool but with a left part taking an argument in Set instead of Prop. Following the definition of succ, we get:

```
Realizer λk:dim λx:dart
  Fix succ_real {succ_real/1 : fmap → (sumor dart)
  := λm Cases m of
          v ⇒ (inright dart)
        | (i _ mi) ⇒ (succ_real mi)
        | (l (c j z t) ml) ⇒
            if (AND_DEC k j x z)
            then (inleft dart t)
            else (succ_real ml)
     end}
```

As usual, recursion is translated by Fix. The realizer succ_real looks like the one of EXD_DEC, though the syntax of the "output values" is a bit hairy because sumor – parameterized by a Set – is more complicated than sumbool. We use it on darts, so its constructors are (inleft dart) with an argument of type dart, and (inright dart) without argument. They correspond respectively to the left and right part of the specification. AND_DEC is a little decidability lemma easing the proof of EX_SUCC_2.

We have now an interesting extraction yielding successors of darts when they exist, or the information that they cannot be found. This leads to a basic error processing scheme by using Caml exceptions in the latter case.

```
# succ 1 18 m;;
- : Coq.dart = 2
#  succ 0 18 m1;;
Uncaught exception: Failure("semantic error: '18' doesn't have
0-succ in '1/17 17/1 3/20 ... 1-2 3-4 4-3 ... 1 2 3 ... 28 29'
during command 'succ (EX_SUCC_2)'")
```

We make the system informatively report errors by an additional string representing the working map, with minus signs for 0-sewings, division signs for 1-sewings and numbers alone for dart insertion. Again, this representation is just a convenient mapping of the extraction of free maps, thanks to a string parser we do not describe here. It avoids the chore of writting a litany of constructors and parenthesis. In a more sophisticated prototype, we could catch exceptions – which is easy in Caml – and thus implement error recovery.

Predecessors Strictly speaking, the extraction of the theorem below defines a new Caml function. Actually, it is just another view of succ for free.

```
Theorem EX_SUCC_1 : ∀k:dim ∀y:dart ∀m:fmap
  {x:dart | (succ k x m y)} + {nopred k y m}
```

Tails The specification of a tail of dart z in free map m must mention that z exists in m because we have not – rightly – specified what a tail is on free map v, and that m is well-formed because the recursive calls in tl will stop – on tl_i1 – only for quasi-maps. Hence we write:

```
Theorem EX_TL_2 : ∀k:dim ∀m:fmap ∀z:dart
  {z0:dart | (tl k m z z0)} + ({~(exd z m)} + {~(wf m)})
```

Since this theorem and others – e.g. the functionality of tails – are formally demonstrated, the extracted function is not only bound to terminate but moreover to provide a dart that *is the* tail of a dart belonging to a well-formed map.

Csuccessors Relation csucc relying on succ and tl, all the work was done in the previous sections. Thus, EX_CSUCC_2 is stated, proved and extracted like EX_TL_2. A completed predecessor function is obtained alike.

We now own a useful set of Caml functions making a real prototype. The complexity of an extracted function mirrors the complexity of the corresponding realization, but we do not begin a formal study because it is not the point when prototyping. Nevertheless, a practical evaluation has been undertaken [2]. For functional programs obtained from extraction, quite an unusual programming method, the execution response times are surprisingly low. Testing the well-formedness of a mid-size map like, say 5 times m, does not spend more than a second, and observing it with our selectors is immediate. Besides, with an interactive graphical interface, the well-formedness would take no time if verified step-by-step along the building process, which is desirable.

6 Combinatorial Characteristics

The main characteristics of the combinatorial maps – Euler's characteristic and the genus – are linear combination of the number of darts, vertices, edges, faces and components. Counting darts at the proof level is trivial:

DEFINITION 7 (NUMBER OF DARTS). Let nd be the recursive function from free maps to integers computing the number of darts:

```
Fixpoint nd [m:fmap] : Z
  := Cases m of
        v ⇒ '0'
      | (i _ mi) ⇒ '(nd mi)+1'
      | (l _ ml) ⇒ (nd ml)
    end
```

We reuse the Coq library on integers, with natural syntax between quotes. Though always positive, characteristics are not computed naturals because we want to be able to permute freely successor with predecessor, which is not always possible on whole numbers. This greatly simplifies proofs.

Computing, say the number of vertices, is more complicated. One solution is to consider the binary relation $\mathcal{R}$ on darts: "there exists a path of one-sewings from x to y" and to decrement instead of increment, as for nd. Let 0 be the number of vertices in the void free map. Inserting a dart makes that number increase by 1 beause it is a potential vertex: it may constitute later on (the beginning of) a vertex. Then, sewing unrelated darts w.r.t. $\mathcal{R}$ decreases it. On the contrary, sewing darts belonging to some path of one-sewings has no effect because they have already been taken into account.

Fixpoints are preferred to inductive definitions, because we do not intend to center proofs on counting – which presents no difficulties – but on the relation

[2] on a SUN workstation (sparc 10) with 128 M of RAM

described above. In particular, we need not the "smallest set" property. Moreover, fixpoints extract as is, so theorems like EX_SUCC_2 or EX_TL_2 are not necessary.

As vertices and edges are two specializations of the same concept, namely α_j-orbits respectively at dimension 1 and 0, we easily generalize the corresponding relations in "there exists a path of j-sewings", by parameterizing j. We get expve, of type dim→dart→dart→fmap→Prop. Note that we must show at the proof level that expve is reflexive and transitive for our axiomatics to be correct [11]. Two specializations of expve can be defined by instantiating the dimension parameter on 1 or 0. We thus obtain two new relations: the existence of 1-paths and the existence of 0-paths, of type dart→dart→fmap→Prop, respectively denoted expe and expv. As expv is a projection of expve, the realization of EXPV_DEC is the projection of the proof of EXPVE_DEC at the dimension one.

From these decidability theorems and the explanations given above, we get:

DEFINITION 8 (NUMBER OF VERTICES).

```
Fixpoint nv [m:fmap] : Z
  := Cases m of
       v ⇒ '0'
     | (i _ mi) ⇒ '(nv mi)+1'
     | (l (c zero _ _) m0) ⇒ (nv m0)
     | (l (c one x y) ml) ⇒ Cases (EXPV_DEC y x ml) of
                              (left _) ⇒ (nv ml)
                            | (right _) ⇒ '(nv ml)-1'
                            end
  end
```

Note the case analysis on EXPV_DEC. Execution time can be reduced when testing if we realize that it is useless to call on EXPV_DEC when y is equal to x. Indeed, we do know then that there exists a one-path since expv has been proved reflexive. This is one of the great features of formal program development: though an important modification has been made, we need not bother with either the termination or the correctness of the extraction. Indeed, thanks to the underlying axiomatics and its mechanically proved theorems, we are convinced of the legitimacy of this modification. Fixpoint ne, of type fmap→Z, is defined alike.

We now know how to enumerate darts, vertices and edges. Components and faces are treated in the same lines [11]. Then, we can compute Euler's characteristic and the genus, infer a planarity criterion and prove Euler's formula, which puts an end to our axiomatization. Because those long and difficult developments principally focus on proofs, new extraction concepts are not brought into play.

7 Conclusions

We have proposed a three-stage hierarchical formal specification of the combinatorial maps in CIC. The proofs in Coq of logical properties meeting the requirements of our mathematical model have shown the correctness of the specifications. Thanks to the extraction process, a realistic prototype has been

automatically obtained. The whole development makes a full-scale case-study with its 232 theorems, 40 definitions and more than 10000 proof lines. The extracted part is much smaller: about 500 lines for the raw extraction and twice as more for the various mapping and error processing functions.

Technical formal specification problems – like the handling of hierarchized ordered sorts, the smooth integration of subtypes defined by invariants, and the completion of partial relations – have been addressed at the proof level and from the prototyping viewpoint. This has resulted in a methodology of specification, proof and extraction, opening up new prospects in formal geometric modeling such as the proof of a discrete Jordan's theorem, achieved to date.

The extension of Euler's formula to the third dimension is an open problem that could surely benefit from a systematic formalization in the spirit of this paper. Concerning prototyping, we plan in the future to develop a graphical interface on top of our extracted functions, thus getting a formal map modeler.

References

1. J.A. BERGSTRA, J. HERIG, et al. *Algebraic Specification.* ACM Press, 1988.
2. Y. BERTRAND and J.-F. DUFOURD. Algebraic Specification of a 3D-Modeler Based on Hypermaps. *Graphical Models and Image Processing*, 56(1):29 – 60, 1994.
3. J.-F. DUFOURD. Algebras and Formal Specifications in Geometric Modeling. In *The Visual Computer*, pages 131 – 154. Spinger-Verlag, 1997.
4. J.A. GOGUEN et al. Introducing OBJ3. In *Applications of Algebraic Specifications Using OBJ.* Cambridge University Press, 1992.
5. F. PUITG and J.-F. DUFOURD. Formal Specifications and Theorem Proving Breakthroughs in Geometric Modelling. In *Theorem Proving in Higher Order Logics*, Canberra (Australia), September 1998. Springer-Verlag, LNCS. http://dpt-info.u-strasbg.fr/~puitg/tphol98/main.html.
6. P. LIENHARDT. Topological Models for Boundary Representation : A Survey. In *Computer Aided Design*, volume 23(1), pages 59 – 81. 1991.
7. A. JACQUES. Constellations et Graphes Topologiques. In *Combinatorial Theory and Applications*, pages 657 – 673. Budapest (Hungary), 1970.
8. W. TUTTE. Combinatorial Oriented Maps. In *Canadian Journal of Mathematics*, volume XXXI(2), pages 986 – 1004. 1979.
9. A. JACQUES. Sur le Genre d'une Paire de Substitutions. *Notes des membres et correspondants,* volume 267, pages 625 – 627, 1968.
10. R. CORI. Un Code pour les Graphes Planaires et ses Applications. *Socit Math. de France, Paris*, Astérisque 27, 1975.
11. F. PUITG and J.-F. DUFOURD. Combinatorial Maps and Planarity: Formal Specifications and Proofs in the Calculus of Inductive Constructions. Tech. Rep. 98/05 (100 pages), LSIIT, Strasbourg (France), 1998. http://dpt-info.u-strasbg.fr/~puitg/rr98/main.html.
12. C. PAULIN-MOHRING. Inductive Definitions in the System Coq — Rules and Properties. In *Typed Lambda-calculi and Applications.* LNCS 664, 1993.
13. C. CORNES et al. *The Coq Proof Assistant Reference Manual V6.1.* INRIA-Rocquencourt, CNRS-ENS Lyon (France), December 1996.
14. C. PARENT. Synthesizing Proofs from Programs in the Calculus of Inductive Constructions. In *Mathematics of Program Construction.* LNCS 947, 1995.

Design of Distributed Multimedia Applications (DAMD)

Wanderley Lopes de Souza
Paulo Nazareno Maia Sampaio

GSDR/DC/CCT/UFSCar
P.O. Box 676
13565-905 São Carlos, SP, Brazil
{desouza, psampaio@dc.ufscar.br}

Jean-Marie Farines
Roberto Milton Scheffel

LCMI/DAS/UFSC
P.O. Box 476
88040-900 Florianópolis, SC, Brazil
{farines, roberto@lcmi.ufsc.br}

Maria Janilce B. Almeida
Luciano Paschoal Gaspary
Lisandro Zambenedetti Granville

INF/UFRGS
P.O. Box 15064
91511-970 Porto Alegre, RS, Brazil
{janilce, paschoal, granville@inf.ufrgs.br}

Roberto Willrich
Murilo S. de Camargo
Marcelo Domingos

INE/UFSC
P.O. Box 476
88040-900 Florianópolis, SC, Brazil
{willrich, murilo, domingos}@ine.ufsc.br

Abstract

Design of Distributed Multimedia Applications (DAMD) is a multi-institutional co-operative project aiming the development of a methodology, based on the Formal Description Technique (FDT) *Enhancements to Language of Temporal Ordering Specification (E-LOTOS)* and supported by a set of appropriate tools, for the specification, validation, implementation, and testing of distributed multimedia applications. This paper presents the main results of this project.

1. Introduction

Specification and validation, including verification, simulation and testing, are of crucial importance in the development of distributed multimedia applications. The use of Formal Description Techniques (FDTs) is fundamental to a coherent treatment on the life cycle of these applications. Nowadays, *Extended State Transition Language (Estelle)* [1] and *Language of Temporal Ordering Specification (LOTOS)* [2], standardised by the *International Organisation for Standardisation* (ISO), and *Specification and Description Language (SDL)* [3], standardised by the *International Telecommunication Union-Telecommunication* (ITU-T), are the most common FDTs applied in the design of distributed systems and protocols. Concerning applications in the area of multimedia, limitations have been identified on these FDTs, such as the lack of constructions to represent temporal constraints.

Enhancements to Language of Temporal Ordering Specification (E-LOTOS) [4] is a FDT that has been developed by ISO and aims to enhance the FDT LOTOS. Some E-LOTOS characteristics are : modularity, including module and interface definitions; introduction of a temporal semantic; re-definition of abstract and concrete data types; generalisation of existing

operators (e.g., sequential composition and parallel operator); introduction of exceptions and exception handling; definition of typed gates and partial synchronisation.

Design of Distributed Multimedia Applications (DAMD) is a multi-institutional co-operative project, supported by "*National Council for Scientific and Technological Development - CNPq*" (ProTeM-CC Phase 3 program), involving researchers, technicians and students from the Federal Universities of São Carlos (UFSCar), Santa Catarina (UFSC), Rio Grande do Sul (UFRGS), Espírito Santo (UFES), and the University of Twente (the Netherlands).

The main goal of the DAMD project is the development of a methodology, based on the FDT E-LOTOS and supported by a set of integrated tools, for the specification, validation, implementation, and testing of distributed multimedia applications.

2. DAMD Methodology

During the design of multimedia and hypermedia applications, three structures must be considered [5]: (1) the *conceptual level*, which describes the application components and their logical and temporal relationships; (2) the *presentation level*, which describes the application and its component presentation characteristics; and (3) the *content level*, which describes the information for data access and manipulation. Besides these structures, it is also necessary to consider the possible user interaction methods.

One of the main aspects discussed during the multimedia/hypermedia authoring process is the specification of the temporal and logical behaviour of these applications. Thus, attention was focused on the synchronisation issues of these applications in the first phase of the DAMD project.

Within the context of the DAMD project, an appropriate specification style was developed based on the approach proposed by Courtiat [6] to specify temporal and logical synchronisation of multimedia and hypermedia applications in E-LOTOS [7, 8]. This specification style also enables the complete representation of the three structures presented formerly.

In order to express the synchronisation requirements of a multimedia application, two libraries, the Presentation and Constraint Libraries, were proposed in line with the adopted specification style. The models proposed by Karmouch [9], Buchanan [10], and Sénac [11] were applied to these libraries.

To illustrate the developed specification style, let us consider an interactive multimedia application consisting of two scenes (*Scene1* and *Scene2*). Initially, Scene1, depicted in Figure 1, simultaneously presents: (1) a synchronised audio and video composition (*AV*) using the firing rule *WaitEarliest*, which determines that the end of the compound presentation is driven by the end of presentation of the earliest medium; (2) an image (*Img1*), and (3) an interactive object (*Quit*).

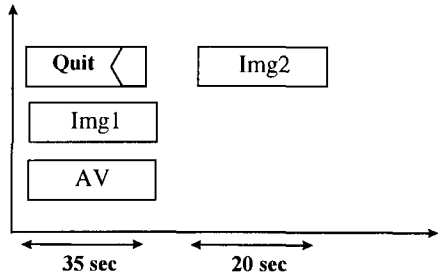


Figure 1. Informal presentation of *Scene1*

The presentation of *AV*, *Img1* and *Quit* is synchronised by the firing rule *WaitMaster*, which determines that the end of the compound presentation is driven by the end of presentation of a medium identified as master (*Quit*). When the *Quit* button is selected or its presentation duration elapses (35 seconds), the media presented simultaneously are interrupted and another image (*Img2*) is presented. When *Img2* presentation duration elapses (20 seconds), *Scene1* is interrupted and *Scene2* is presented.

It is possible to demonstrate how compound presentations can be refined into the composition of presentation objects (*Quit*, *Img1*, *AV* and *Img2*) and possibly synchronised by constraint objects (*WaitMaster*), through the graphical representation of *Scene1*.

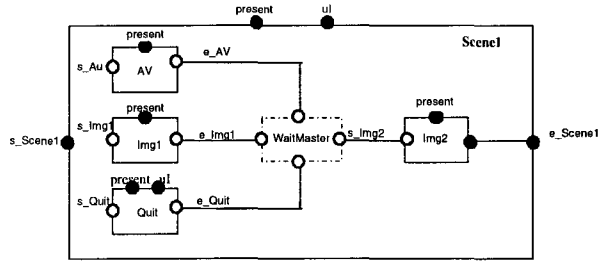


Figure 2. E-LOTOS architecture of *Scene1*

Figure 2 illustrates the E-LOTOS architecture of *Scene1*. The events *s_Scene1* and *e_Scene1* denote, respectively, the initial and final events of *Scene1*. The *e_AV*, *e_Img1*, *e_Quit* and *s_Img2* events express the implicit synchronisation among the scene components. The presentation and content characteristics of the application are offered to its environment by the *present* gate. Finally, the user interactions are accomplished through the *ul* gate.

Figure 3 presents the E-LOTOS architecture of *AV*, which is refined into the parallel composition of the processes *Audio* e *Video*. The end of this composition is driven by the constraint process *WaitEarliest* through the final events e_Audio and e_Video.

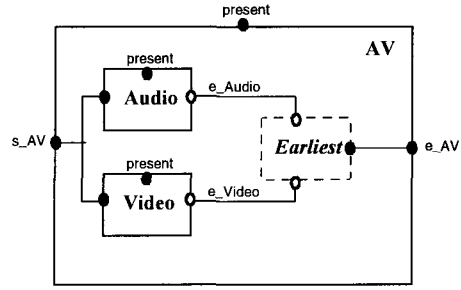


Figura 3. – E-LOTOS architecture of *AV*

Appendix A illustrates an overview of the E-LOTOS specification for the aforementioned application.

3. DAMD Tool Set

The integrated DAMD Tool Set (Figure 4) is constituted of : specification generation tools (E-LOTOS Textual Editor, E-LOTOS Graphical Editor and Authoring Language Editor), validation tools (Verification and Simulation Tools), and implementation tools (MHEG-5 and JAVA Translators).

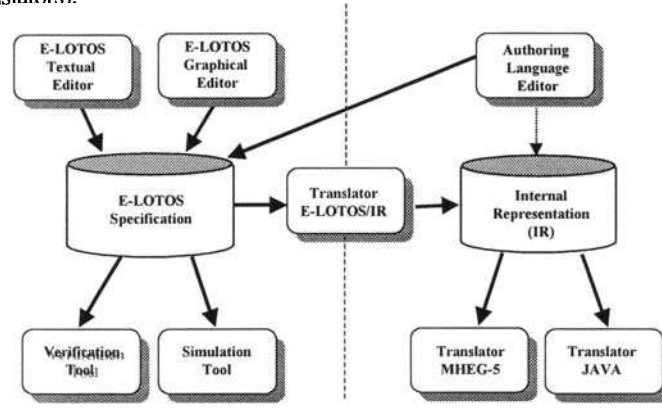


Figure 4. DAMD Tool Set

The E-LOTOS Graphical Editor allows an E-LOTOS expert to specify his application by means of a visual language. The Authoring Language Editor allows an E-LOTOS non-expert user to specify his application by means of a friendly interface. These tools generate an equivalent E-LOTOS textual specification. The user then proceeds with the verification and simulation phases, where correctness of the specification behaviour can be verified.

After validating the E-LOTOS specification, an Internal Representation (IR) [12], which enables the integration of the whole DAMD tool set, must be produced to provide the implementation of the multimedia application. All the components of a multimedia application identified in an E-LOTOS specification can be expressed on the IR. These components are : (1) *Application*, which encapsulates all the other application components and triggers the first

scene to be presented; (2) *Scene*, which is constituted of their components (*Presentation* and *Composition*) and presentation attributes, denotes a particular context to be presented within an application; (3) *Composition*, which is composed of monomedia objects (e.g., audio, text, etc.), other possible compositions and their presentation attributes, represents a grouping of information associated with a logical or temporal synchronisation; (4) *Presentation*, which describes the information to be presented and interactive objects (e.g., buttons) for navigation control in the application.

Finally, the IR is translated to a concrete representation (MHEG-5 or JAVA), so that the validated specification can be implemented in a distributed environment.

4. Specification Tools

4.1. E-LOTOS Graphical Editor

The E-LOTOS graphical editor is an extension of the DART graphical syntax [13], and is composed of: the new graphic syntax E-DART, an editor that supports this syntax (E-DART Editor), and a translator to convert the graphical diagrams to the textual syntax of E-LOTOS. Figure 5 illustrates the graphical interface of this tool.

Modules, interfaces and generic modules are E-LOTOS elements with no graphical representation. The editor supports such constructions through a specification tree, which organises the specification into three main elements. Each module presents folders for the data types, functions and processes. The proposed syntax focuses on the graphical specification of the system behaviour. Therefore, data types and functions are still specified textually.

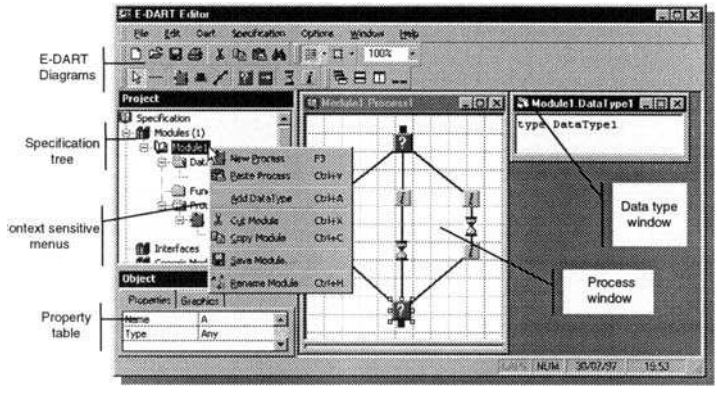


Figure 5. E-DART graphical interface

The system processes can also be accessed through the specification tree. Each process has a graphical window for the specification of its own dynamic behaviour. The gates, as well as the parameters, are created and managed in each process sub-tree. The process also has a table of properties with its name and the return type. The definition of the dynamic behaviour of a

process is made through the insertion of graphical symbols into the process window. Like the process, each of these diagrams also has a table of corresponding properties.

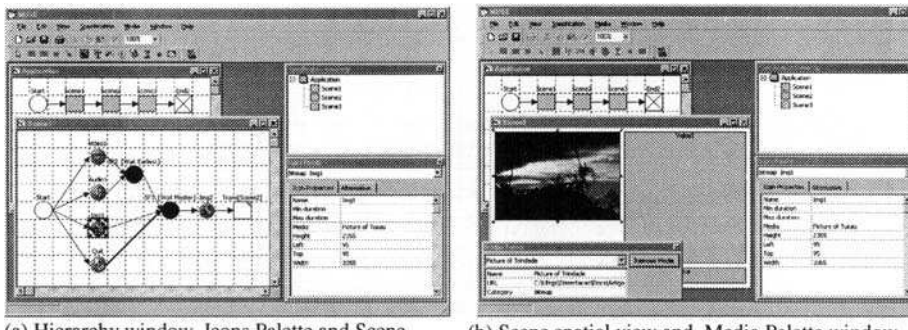
The E-DART editor functions are expressed by user interactions through context-sensitive menus. The specification of a system is done by the definition of the system modules. Each new module is defined as a folder into the specification tree. New data types, functions and processes can be created in this folder. For each process created, a new sub-tree is also presented that enables the user to specify gates, parameters and local variables of the process.

Modules already defined by other specifications can be reused. This facilitates the construction of new specifications, in addition to increasing their reliability, since they are based on components already tested. This tool still generates the equivalent textual E-LOTOS specification at the end of the editing process.

4.2. Authoring Language Tool

This tool allows for the specification of multimedia applications according to an authoring model that couples mechanisms for logical structuring of the applications to a synchronisation model similar to HTSPN [14]. The logical structuring level is based on the concept of scenes and groups, providing a broad view of the application. The definition of temporal synchronisation is done in each scene by means of a simplified graph. Spatial synchronisation allows media objects to be positioned according to the output device. A detailed description of this model can be found in [15] and [16].

The creation environment is divided into two units: media objects repository and specification area. The user can insert media objects into the repository at any given moment. This is done by browsing a local media object or referencing a remote one. Figure 6b illustrates the Media palette window, which is used to manage the media objects referenced by the application.



(a) Hierarchy window, Icons Palette and Scene temporal view

(b) Scene spatial view and Media Palette window

Figure 6. Interface of the authoring environment

The specification area is composed of scenes and groups. Each scene is represented by both the temporal and the spatial views. The specification area allows the user to insert icons and synchronisation points and to establish their relationship using arcs. The visible elements used in the temporal synchronisation can be adjusted in the spatial view.

Figure 6a illustrates the basic functionality of the authoring environment. The toolbar has shortcuts to its main functions. Two support windows can be observed: Specification hierarchy and Icons palette. The former provides the user with a general view of the application, presenting all the scenes and groups in a tree and providing a structured view of the relationships among them. In this case, the application modelled in section 2 is composed of two scenes: *Scene1* and *Scene2*. The latter, in turn, provides mechanisms to visualise and edit the icons' properties. In the same figure, the bitmap icon (*Img1*) of *Scene1* is selected and its specific properties are presented in the aforementioned window. Icons that have an associated media object (audio, text, image, and video) present a special property called *media*. This property must be filled with a media object existing in the repository. In this example, icon *Img1* is associated with the media object Picture of Trindade. Figure 6b also illustrates the spatial view of *Scene1*, where it is possible to move or resize the visible media objects. Their properties related to co-ordinates and dimensions are automatically updated in the Icons palette.

This tool also provides means to reuse scenes and groups. Finally, it is also important to highlight the functionality of E-LOTOS and IR code generation, which is obtained through the special saving options *Save as E-LOTOS* and *Save as IR*, respectively.

5. Validation Tools

5.1. Verification

Verification of properties in multimedia applications corresponds, as in another systems, to verifying two main groups of properties: *general properties*, which are independent of its characteristics, and *specific properties*, issued from the particular features of the application. The general properties to be verified are *liveliness* and *safety*. In a multimedia application, the property of liveliness means that all the media will be presented at some time during the presentation phase. The safety property means that the system never reaches a state (also called deadlock) which doesn't allow its termination. The specific properties should be identified for each application based on the logical and temporal relationships among its components.

Verification of the general and specific properties provides correctness during the system's development. However, some media relationships, mainly temporal relationships, are strongly affected by the execution platform. Consequently, platform characteristics also have to be considered when the properties must be fulfilled (e.g., conflicts of access, lack of resources for the presentation, etc.).

The verification process consists of translation of the system's behaviour from the E-LOTOS specification to a *timed automaton*. Verification of properties is then performed over this automaton.

The verification tool uses the *Hybrid Technology (HyTech)* tool [17], a symbolic model checker based on an analysis approach for linear hybrid automata, presented in [18] and [19]. Timed automata is considered a particular case of this general model.

The HyTech approach uses the transformation predicate for the computation of the predecessor and successor regions of an initial region. A *region* is a group of the automaton's states which corresponds to the automaton's conditions and values of the variables. The properties to be verified are described using temporal logic and the regions by the Hytech tool. The HyTech tool also accomplishes the related parametric analysis, substituting some values of variables for symbolic parameters. When possible, the tool calculates the values of the symbolic parameters whose properties are fulfilled, based on the formula that describes these properties.

The architecture of the verification tool is presented in Figure 7. After the timed automaton generation from E-LOTOS, the verification commands the general properties to be automatically generated and submitted to the HyTech tool. The user must then determine the specific properties to be verified, translating them into initial and final regions. The necessary commands to be used in HyTech are also generated. Also, the user should determine the devices for the media presentation which are available on the platform to be employed. Verification of the platform, as well as parametric analysis, is done using HyTech commands. In the case of parametric analysis, the user must inform only the parameters to be calculated.

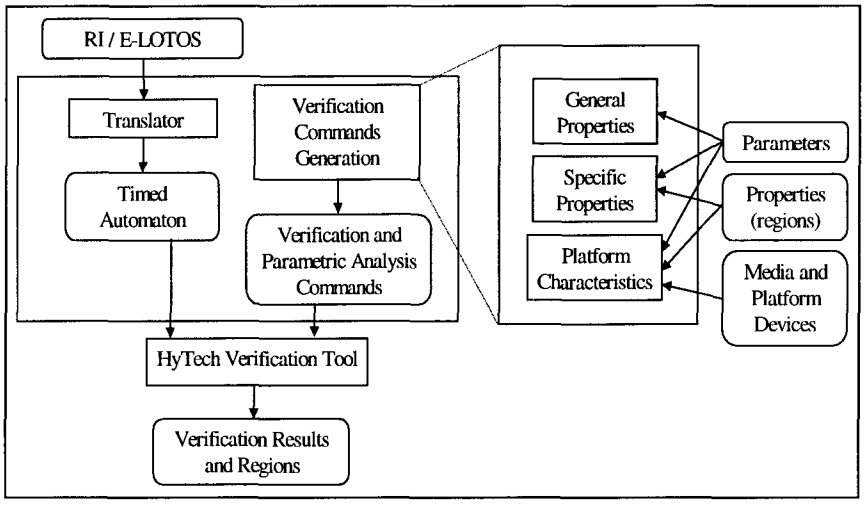


Figure 7. Architecture of the verification tool.

5.2. Simulation

The main functions provided by the *E-LOTOS Simulator (ELS)* are : (1) *Interactive Simulation*, (2) *Snapshots*, and (3) *Execution Traces*. *Interactive Simulation* enables to choose an action to perform a system evolution from a list of the current enabled actions at any time (e.g., time progress actions). Execution specification *snapshots* can be produced at any execution state. These snapshots provide an overview of the specification execution. Based on *execution traces* all performed actions can be automatically recorded in a log file, which can be executed afterwards. Traces can be pre-designed in order to make a broad exploration of the

possibilities of evolution. A trace file allows for the production of snapshots and verification of any specification state within the trace.

In order to accomplish the simulation using ELS, the E-LOTOS specification is translated into an equivalent internal tree representation from which the simulation is carried out. An essential element of the simulation tree is called behaviour. Behaviours can be composed of gates, sequential composition, choice, interleave, synchronisation, etc. Each behaviour can be preceded and/or succeeded by expressions, which can be either an assignment execution or a sequential composition of assignment expressions. All the expressions preceding a behaviour in a specification are considered as the succeeding expression of the previous behaviour. Whenever such preceding behaviour does not exist, the expression is the one preceding the actual behaviour. If there is no behaviour associated with the expression, then it is associated with the special behaviour *exit*.

The simulation tree is composed of objects. All the objects in the tree are descendants of the specification behaviour. The leaf edges of this tree are either gates or process instantiations. The process instantiation is produced by adding its associated behaviour to the tree. When the instantiation is concluded, its behaviour is removed from the tree. When a gate is enabled it is added in the enabled actions list.

The simulation tree allows to answer queries such as which gates are enabled. Each behaviour class has the knowledge about the operator that it implements. Each class can be queried about which gates are enabled in a given execution moment, and these gates answer according to their knowledge. Thus, to obtain the list of enabled gates it is only necessary to ask about it to the root of the simulation tree.

Data expressions are defined by an E-LOTOS sub-set and interpreted by ELS using operator prefixing. Each operator has a number of pre-defined parameters. Each parameter can be a data type value, a variable or an expression. This composition defines a tree structure, where the variables are bound to the process/function in execution.

6. Implementation Tools

6.1. Translator to MHEG-5

This tool has been implemented based on a methodology previously defined within the context of the DAMD project [20]. The translation from IR to *Multimedia and Hypermedia Information Coding Expert Group* – Part 5 *(MHEG-5)* [21] is enabled by the correspondence between their representations, and is accomplished by the implementation of procedures to automatically translate the application components, their logical and temporal relationships and their content and presentation characteristics.

An MHEG-5 application is described by the declaration of two objects : (1) *Application*, which defines the application to be presented and (2) *Scene*, which characterises the multimedia scenarios of the application and its components. In order to present an MHEG-5 application, the former must be described by only one *Application* object and at least one

Scene object. Thus, the IR/MHEG-5 translation is oriented to the generation of the MHEG-5 *Application* and *Scene* objects.

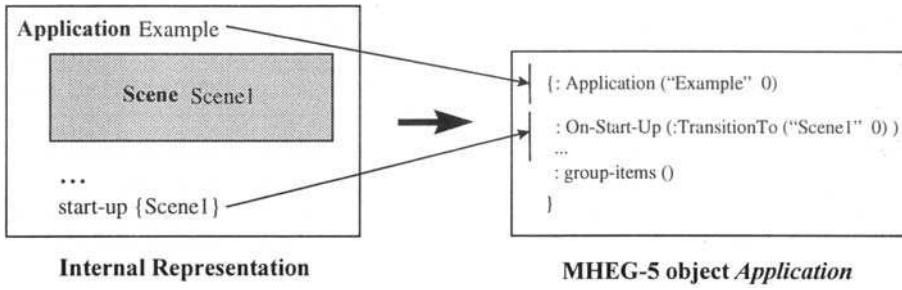


Figure 8. Translation of IR *Application* component to MHEG-5

Figure 8 illustrates the translation from IR to MHEG-5 of the multimedia application presented in section 2. The *Application* object is obtained based on the IR component *Application*, where the application name, its components (*Scenes*) and the first scene to be presented (*start-up* attribute) are declared.

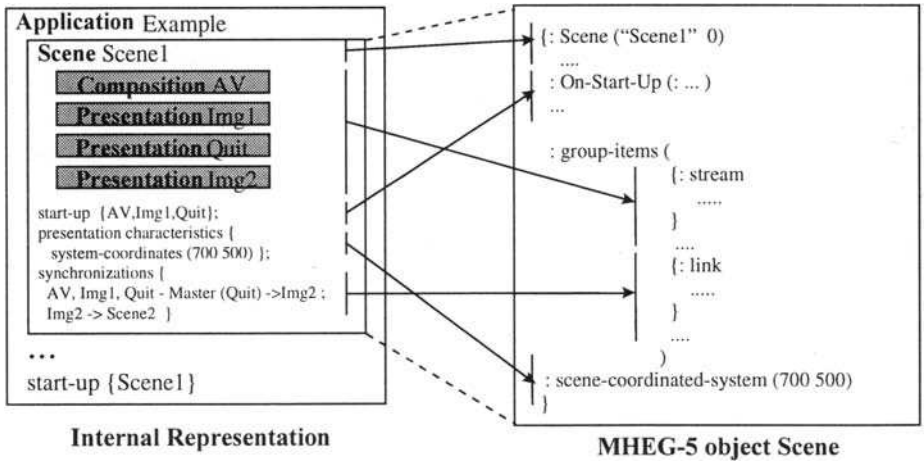


Figure 9. Translation of IR *Scene* component to MHEG-5

The *Scene* object is obtained based on the IR component *Scene* (Figure 9). During this translation, the application's scenes are mapped to MHEG-5 individually, enabling representation of the scenes' names, the media to be initially presented, their components (*compositions* and *presentations*), their logical and temporal synchronisation and their presentation characteristics.

6.2. Translator to JAVA

In order to complete the authoring process of multimedia applications, a tool was developed to provide automatic generation of Java applications based on the validated specification of multimedia applications. All generated Java applications are composed of two classes: *Application* and *Interpretation* classes.

The *Application class* is automatically generated from the Internal Representation (IR) describing the multimedia application . This class is the starting point for the Java multimedia applications and contains IR as its attribute. When interpreted, an object of this class instantiates an active object (implemented in Java by a thread) of the Interpreter class (presented below).

The Interpretation classes are responsible for the presentation of any application described according to the authoring language proposed in this paper. Among these classes, the *Interpreter* class is responsible for the creation and control of an *Interpretation Structure*. This structure is composed of a set of active and communicating objects. These objects implement all the characteristics specified by the Internal Representation.

The Interpretation Structure of a scene[1] or of a generic group G is composed of a set of presentation modules representing the presentation of components and one scheduler module.

There are six presentation modules: *PrText*, *PrImage*, *PrAudio*, *PrAnimation*, *PrVideo*, and *PrButton*. Each presentation module is composed of three active objects: *Presentation*, responsible for presentation of the component; *TV-Control*, responsible for the control of the Temporal Validity (TV) of the component presentation; and *Alternative-Presentation*, responsible for change of the component content if the main content cannot be presented (except of PrButton).

The Scheduler module is responsible for the presentation scheduling modelled by the components of a generic group G and is composed of two active objects: *Player*, responsible for the evolution of the multimedia scenario modelled by group G; *TV-Control*, responsible for the control of the temporal validity of G.

During the creation of the interpretation structure associated with a group G, the Interpreter is responsible for linking a Scheduler module to the active objects that represent the components of G (it could be other Scheduler modules). From these links, the Scheduler module receives state changes of components and issues stop and start interpretation commands.

Figure 10, for instance, illustrates part of the structure necessary for the interpretation of the application presented earlier herein (section 2). The application is represented by a Group module called Application (Figure 10a). Figure 10b presents refinement of the Application

[1] A scene can be considered as a group formed by one scene.

module. It is composed of a Scheduler module and two Group modules. Figure 10c and 10d refine the *Scene1* and *AV* Group modules, respectively.

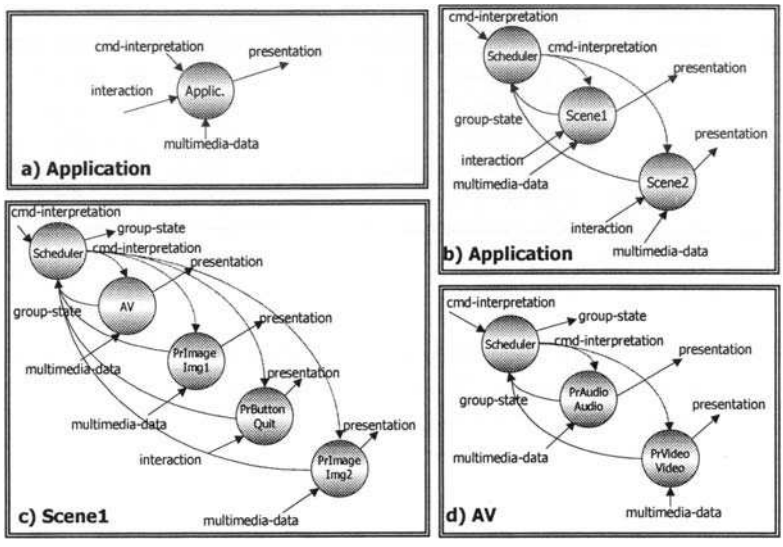


Figure 10. Example of interpretation structure

7. Conclusions

This paper presented the methodology and set of tools developed in the first phase of the DAMD project. In this phase, the major concern was the synchronisation problems. Thus, the *Presentation* and *Constraint* libraries were constructed based on several synchronisation models [9, 10, 11] to express the temporal and logical behaviour of a multimedia application. An author is able to specify interactive multimedia applications through the use of these libraries and a well-defined E-LOTOS specification style [22].

In the first phase of DAMD project three kinds of tools were developed: specification tools, validation tools, and implementation tools. The integration of these tools was accomplished through the proposal of an Internal Representation (IR), and the evaluation of the DAMD methodology and set of tools could be verified by the development of several interactive multimedia applications.

The second phase of this project will focus on investigation of the problems derived from distribution and Quality of Service (QoS). Some applications have been already studied and the *Presentation* and *Constraint* libraries will probably be extended. As a consequence, these extensions will be incorporated in the DAMD Tool Set.

References

[1] ISO/IEC 9074, *Information Processing Systems - Open System Interconnection - Estelle - A Formal Description Technique Based on an Extended State Transition Model*, 1989.

[2] ISO/IEC DP 8807, LOTOS - *A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, 1987.

[3] Z.100, CCITT Blue Book Vol. X, Fasc.X.1, ITU Geneva, 1993.

[4] ISO/IEC JTC1/SC21/WG7, *Working draft on enhancements to LOTOS*, Project WI1.21.20.2.3, February, 1998.

[5] R. Willrich. *Formal Design of Portable Hypermedia Documents* (in French). PHD Thesis, LAAS-CNRS, Université Paul Sabatier. Toulouse (France), September ,1996.

[6] J.-P. Courtiat, R.C. de Oliveira. *Proving Temporal Consistency in a New Multimedia Synchronisation Model*, in ACM Multimedia, Boston, MA, USA, 1996.

[7] P.N.M.Sampaio, C.Y.Shiga & W.L.deSouza. *Enabling Temporal Synchronisation in E-LOTOS Specifications and MHEG-5 Applications*, proceedings of IEEE Conference on Protocols for Multimedia Systems - Multimedia Networking (PROMSMmNet'97), Santiago-Chile, November, 1997.

[8] P.N.M.Sampaio & W.L.deSouza. *Modelling Multimedia and Hypermedia Applications using an E-LOTOS/MHEG-5 Approach*, proceedings of Workshop on Conceptual Modelling in Multimedia Information Seeking, Los Angeles-USA, November, 1997.

[9] N. Hirzalla, B. Falchuk & A. Karmouch. *A Temporal Model for Interactive Multimedia Scenarios*, IEEE Multimedia (2) N° 3, pp 24-31, 1995.

[10] M.C. Buchanan & P.T. Zellweger. *Automatically Generating Consistent Schedules for Multimedia Documents*, Multimedia Systems Journal, 1(2):55-67, 1993.

[11] P. Sénac, M. Diaz, A. Leger, & P. de Saqui-Sannes. *Modelling Logical and Temporal Synchronisation in Hypermedia Systems*, IEEE Journal on Selected Areas in Communications, 14(1):84-103, January, 1996.

[12] P.N.M.Sampaio & R.Willrich. *Internal Representation for Interactive Multimedia Applications* (in Portuguese), Internal report – DAMD Project, October, 1997.

[13] J.S. Allende, *Graphical Designer for LOTOS*, User Manual, Dept. of Telematic Systems Engineering, Technical University of Madrid, May, 1997.

[14] P. Sénac, M. Diaz, A. Léger & P. de Saqui-Sannes. *Modelling Logical and Temporal Synchronisation in Hypermedia Systems*, IEEE Journal on Selected Areas in Communications, 14(1): 84-103 , January, 1995.

[15] L. P. Gaspary & M. J. Almeida. *Authoring and E-LOTOS Conception of Interactive Networked Multimedia Applications in MUSE Environment*, proceedings of PICS'98-Performance of Information and Communications Systems, Lund, Sweden, May 1998.

[16] L. P. Gaspary & M. J. Almeida. *MUSE - An Interactive Networked Multimedia Applications Specification Environment with E-LOTOS Translator*, proceedings of CAiSE'98-10th Conference on Advanced Information Systems Engineering, pp. 273-287, Pisa, Italy, June 1998.

[17] Henzinger, Thomas A., Ho, Pei-Hsin., Howard, Wong-Toi. *HyTech: The Next Generation*, Proceedings of the 16th Annual IEEE Real-time Systems Symposium (RTSS 95), IEEE Computer Society Press, 1995, pp. 56-65.

[18] R. Alur, C. Courcoubetis, T. Henzinger, P. Ho, X. Nicollin, A. Olivero & S. Yovine. *The algorithmic analysis of hybrid systems*, in Proceedings of the 11th International Conference on Analysis and Optimization of Discrete Event Systems, pg. 331-351. Springer Verlag, 1994.

[19] S. Yovine. *Methods and Tools for the Symbolic Verification of Temporal Systems* (in French), PhD Thesis, Institut National Polytechnique de Grenoble, 1993.

[20] P.N.M.Sampaio. *A Methodology for the MHEG-5 Implementation of Interactive Multimedia Applications Specified in E-LOTOS* (in Portuguese), M.Sc. dissertation , PPG-CC/UFSCar, Brazil, February, 1998.

[21] ISO/IEC IS 13522-5. *Information Technology - Coding of Multimedia and Hypermedia Information : Part 5 : Support for Base-Level Interactive Applications*,1996.

[22] R.M.Scheffel, P.N.M.Sampaio, C.Y.Shiga & R.Willrich. *Specifying Interactive Multimedia Applications using E-LOTOS* (in Portuguese). Internal report – DAMD project, October, 1997.

Appendix A. Example of an E-LOTOS specification of a multimedia application

```
module Scenes import Components, Constraints, Disrupt, Classes is
process Scene1 [s_Scene1, e_Scene1, ul, present:class] (...) : exit is
                hide s_AV, e_AV, s_Img1, e_Img1, s_Quit, e_Quit, req_disrupt,
                     req_end in
                loop forever in
                        (s_Scene1; present (!data) ;
                        ( loop forever in
                                ( (     (     AV [s_AV, e_AV, present] (...)
                                        ||| Img1 [s_Img1, e_Img1, present] (...)
                                        ||| Quit [s_Quit, e_Quit, ul, present] (...) )
        |[e_AV, e_Img1, e_Quit]|
                                cMaster [e_AV, e_Img1, e_Quit, req_end] ) [> Disrupt[s_Img2] )
        |[req_end, s_Img2]|   Request_Disrupt[req_end, s_Img2]  )
        |[s_Img2]|                      Img2 [s_Img2, req_disrupt, present] (...)
                        endloop.
                        [> Disrupt[e_Scene1] )
      |[req_disrupt, e_Scene1]| Request_Disrupt[req_disrupt, e_Scene1]  )
                endloop.
endproc.
endmod.
module Components import Classes, Presentation is   (* Scenes' components declaration *)
process AV [ s_AV, e_AV, present : class ] (...) : exit is
        hide e_Au, e_Video in
                ( (  Au [ s_AV, e_Au, present] (...)     |[s_AV]|
                     pV [ s_AV, e_Video, present] (...) )
                |[ e_Au, e_Video ]|
                   cEarliest [ e_Au, e_Video, e_AV ]  )
endproc
process Au [s_Au, e_Au, present:class] (vAu:StreamClass, dAu:time) : exit is
        pSsSe [s_Au, e_Au, present] (vAu, dAu)
endproc.
...
endmod.
module classes is ... endmod.                    (* data types declaration *)
module Presentation is                           (* Presentation library declaration *)
Process     pSsSe[start, end, present:class] (data : class, d : time) : exit is
            start; present(!data); wait(d); end@!0 ; exit
endproc
...
endmod.
module Constraints is          ... endmod.              (* Constraint library declaration *)
module Disrupt is ... endmod.
specification Example    import ... in [s_app, e_app, ul, present:class]
                behavior
                local
                var vScene:SceneClass, vImg1:StreamClass, ...
                init
                    ?vScene := (scene_coordinated_system => (800 600),  ...          ) ...
                in
                ( (  par s_Scene2#2
                     [s_Scene2] -> Scene1 [s_Scene1, s_Scene2, ul, present](...)
                  ||  [s_Scene2] -> Scene2 [s_Scene2, e_Scene2, ul, present](...)
                     endpar    ) [> disrupt [e_app]  )
                |[e_Scene2, e_app]|   resquest_disrupt [ e_Scene2, e_app]
endspec.
```

Structured Formal Verification of a Fragment of the IBM S/390 Clock Chip

Alfons GESER[1]* and Wolfgang KÜCHLIN[1]

Wilhelm-Schickard-Institut für Informatik, Universität Tübingen

Abstract. We present a simple and powerful method for formal verification of hardware that exploits hardware symmetries. We illustrate the method at an industrial example: a fragment of the IBM S/390 Clock Chip.

List of Keywords: propositional satisfiability, functional decision diagram (FDD), term rewriting, term graph rewriting, formal hardware verification.

1 Introduction

With the ever increasing complexity of hardware, it becomes increasingly important to eliminate logical defects early. Simulation and Formal Verification are two ways to find and analyze logical defects. Simulation is fast, however its runs may be long and plenty of detail irrelevant to the defect. On the other hand, Formal Verification gets to the very problem directly, but may fail in the ambitious attempt to capture the complete behaviour in one effort.

By the fact that hardware is organized in bits, propositional reasoning is a mandatory part of Formal Verification. The now classical approach is *symbolic model checking* [BCL+93]. Model checking determines the set of states of a given finite automaton which satisfy a given temporal or modal formula. The propositional fragment of model checking is decidable, although PSPACE-hard. Symbolic model checking does not operate at the explicitly given state transition relation, which may be as large as 2^{100} nodes, but at a representation of the propositional formula for the relation.

The basic shortcoming of model checking, in our view, is its rigid language and the absence of structure. We pursue therefore another approach: a combination of *term rewriting* and *propositional reasoning*. Term rewriting is a Turing-powerful formalism that is based on the application of rewrite rules $\ell \to r$. A term t in which an instance of ℓ appears may be reduced to a term t' which is like

* Partially supported by grant Ku 966/3-1 of the Deutsche Forschungsgemeinschaft within the Schwerpunkt Deduktion at the University of Tübingen. **Corresponding author.** Address for correspondence: Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Sand 13, D-72076 Tübingen, E-mail: geser@informatik.uni-tuebingen.de

t but the instance of ℓ is replaced by the appropriate instance of r. Term rewriting is easy to learn and comes with a rich body of theory. From a conceptual point of view, propositional reasoning is a part of strongly typed rewriting.

We switched from term rewriting to the related term graph rewriting for its ability to adequately represent hardware structure. The particular progress presented in this paper is a theorem that justifies a complete separation of the propositional reasoning part from the term graph rewriting part.

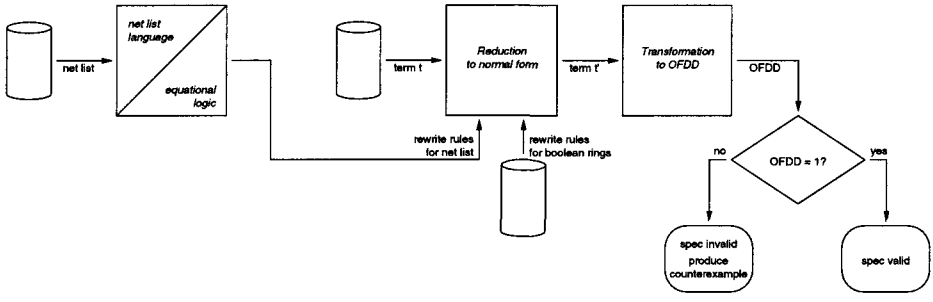


Fig. 1. The translation and reduction pipeline of our approach

In our approach the verification problem breaks down in three steps (see Figure 1):

1. Compilation of the netlist into modules of a typed term graph rewriting system. Sequential behaviour is modelled by deterministic Mealy automata.
2. The formal requirements are rewritten into a normal form, which is a term graph representing a propositional formula.
3. The propositional formula is evaluated as an ordered functional decision diagram. If the result represents 1 then the answer is "yes", else a counterexample may be constructed mechanically on request.

We illustrate these steps at a case study: a fragment of the IBM S/390 Clock Chip.

2 Compilation of Netlists to Term Graph Rewriting Systems

Hardware verification can be done in various abstraction levels: Register-transfer, gate-level, switch-level, and finally on the physical level where at the extreme one has to solve differential equations. See [Gup92] for a survey of hardware verification. Gate-level descriptions model signals by the truth values 0 for "false" and 1 for "true" by which they are convenient both for computer scientists and electrical engineers. We translate gate-level hardware into term graph rewriting systems in which we can reason about hardware behaviour.

2.1 Term Graphs and Term Graph Rewriting Rules

We adopt the operational view of term graph rewriting [SPv93,Cou94] which is closely related to term rewriting [DJ94].

Recall that a *term graph structure* is a set of nodes, a set of arrows between nodes, and a set of function symbols, such that the following properties hold. Each function symbol is assigned a fixed nonnegative integer, its arity. A node may be labelled by a function symbol, otherwise it is called a *variable node*. The number of outgoing arrows from a node must equal 0 for a variable node, and must equal the arity of the label of the node otherwise.

The fact that a node o is labelled by f and has outgoing arrows to nodes $i_1, \ldots, i_n$ may be expressed by an *assignment* $o = f(i_1, \ldots, i_n)$. So a set of assignments forms a term graph structure.

A *term graph* x where G now is a node x together with a term graph structure G; a term graph rewriting rule $\ell \to r$ where G is a pair of nodes ℓ, r together with a term graph structure G.

We consider term graphs equivalent if their structures reachable from the roots x are isomorphic. Likewise term graph rewriting rules are equivalent if their structures reachable from their pairs of roots ℓ, r are isomorphic. Note that we do therefore not care about garbage that may arise during rewriting.

For instance let x, y denote the input nets, and let z denote the output net of some AND gate. If the AND function is denoted by $*$ then the facts are summarized by an assignment $z = (x * y)$.

For instance assume that one wants to model the AND by a two NANDs. This is achieved by a term graph rewriting rule

$$\ell \to r \qquad \text{where} \qquad \ell = (x * y), \quad r = \mathbf{nand}(z, z), \quad z = \mathbf{nand}(x, y)$$

which expresses that the output node ℓ of an AND gate having inputs x, y may be replaced by the output node r of a NAND gate both inputs of which are connected to the output, z, of another NAND gate having inputs x, y.

Terms and term rewriting rules may be considered particular term graphs and term graph rewriting rules, respectively, by assuming a fresh node for each non-variable subterm position.

2.2 Compilation of Combinatorial Hardware

Combinatorial hardware is from the gate-level point of view, all hardware but the storing elements, the flipflops. Abstractly it is a function from a propositional vector, the input vector, to a propositional vector, the output vector.

We adopt the view that nodes in a term graph are typed. Besides its documentation typing provides a means to distinguish between propositions and other term graphs, a fact that we will use in Theorem 1. The notion of type as we will use it is introduced in the *algebraic specification* framework. See e.g. [Wir94].

Let $\mathcal{S}$ and $\mathcal{F}$ be two disjoint sets (the elements of which are called the types and function symbols, respectively), and let $\mathcal{C} \subseteq \mathcal{F}$ denote a set of constructors.

Each function symbol, f, is assigned its *result type*, $\rho(f)$, and its finite tuple of *argument types*, $(\alpha_1(f), \ldots, \alpha_n(f))$, a fact that is occasionally written as $f : \alpha_1(f) \times \cdots \times \alpha_n(f) \to \rho(f)$. A term graph structure is called *typable* if it has a *typing function*, i.e. a function τ that maps nodes to types such that for every assignment $o = f(i_1, \ldots, i_n)$ one has $\tau(o) = \rho(f)$ and $\tau(i_1) = \alpha_1(f), \ldots, \tau(i_n) = \alpha_n(f)$.

Definition 1. *A* typed term graph rewriting system *is a tuple* $(\mathcal{S}, \mathcal{F}, R)$ *where* R *is a set of term graph rewriting rules over* $\mathcal{F}$ *and a set* $\mathcal{X}$ *of variables such that types are preserved: For every rule* $\ell \to r$ where G *one has a typing function* τ *for* G *such that* $\tau(\ell) = \tau(r)$.

We presume fixed the type $\mathbb{B}$ of propositions, complete with the truth values $0, 1$, and the propositional connectives $\neg$, $*$, $+$, $\Rightarrow$, $\Leftrightarrow$ and $\oplus$ which denote NOT, AND, OR, IMPLIES, EQUIV, and XOR, respectively, in descending order of precedence. Let the set of constructors of result type $\mathbb{B}$ be $\mathcal{C}_{\mathbb{B}} = \{0, 1\}$, and let the propositional connectives be function symbols. A propositional formula so is represented by a (typable) term graph the nodes of which are labelled by propositional connectives, propositional variables, and $0, 1$. A specification, on the other hand, is represented by a term graph of type $\mathbb{B}$ which may (and usually will) still contain nodes labelled by various other functions.

For propositional vectors of various lengths $k \geq 0$, let there be types $\mathtt{Bit}_k$ complete with constructors $\mathtt{mk}_k : \mathbb{B}^k \to \mathtt{Bit}_k$ and selectors $\mathtt{sel}_k : \mathbb{N} \times \mathtt{Bit}_k \to \mathbb{B}$ and their rewriting rules

$$\mathtt{sel}_k(n, \mathtt{mk}_k(x_{k-1}, x_{k-2}, \ldots, x_0)) \to x_n$$

for $0 \leq n \leq k - 1$.

By the fact that a combinatorial hardware acts like a function that maps a propositional vector to a propositional vector, the translation from netlists that contain no storing elements to term graph structures is quite natural: A *net* (i.e. a point that carries a signal) in the hardware translates to a node in the term graph; an output net o of a *box* (i.e. a primitive component) whose inputs are the nodes $i_1, \ldots, i_n$ translates to an assignment $o = f(i_1, \ldots, i_n)$ where f is a symbol for the propositional function realized by the box output.

An inner net (i.e. not an input pin) in the hardware may be input to a number of boxes. This "sharing" of intermediate values is typical for hardware. Its ability to carry this over is one of the big advantages of term graph rewriting: as one assignment is created per box output, the size of the created term graph is obviously linear in the size (number of nets) of the hardware. This is in contrast to proper term rewriting where the created term may well be of exponential size. Linearity is most important in view of circuits that may have several hundred nets and flipflops.

2.3 Compilation of Sequential Hardware

To model sequential behaviour of hardware, one has to introduce a form of discrete timing. We adopt the following picture: Every net carries a bit per

unit interval of the clock; the signal is changed at the clock ticks which are assumed instantaneous. One way to express timed behaviour is to assign to a net a function from nonnegative integers (representing time instants) to $\mathbb{B}$ [BK95]. Another is to switch to a modal logic formalism [BCDM86]. We stick to the classic concept of *deterministic finite-state machine* (DFSM), more accurately its *Mealy automaton* variant, to model sequential behaviour of hardware.

Recall that a Mealy automaton is given by three finite sets, S, I, O, whose elements are called states, inputs, and outputs, respectively; an element $s_{\mathtt{init}} \in S$ called the initial state; a function $\delta : S \times I \to S$, the step function; and a function $\lambda : S \times I \to O$, the output function.

We encode states, inputs, and outputs each as propositional vectors. The pins of the chip become the inputs and outputs of the state machine. The storing elements of the chip are D-type flipflops. Assuming all flipflops numbered consecutively, the present state of the chip is given by the enumeration of all nets incident with flipflop Q outputs; the subsequent state by the enumeration of their D inputs. So the state transition function is virtually just another name for the combinatorial part of the state machine, and so easily specified as a term graph rewriting rule.

For instance a 3-bit counter initialized to 0 with preset (enabled by $\mathtt{pe} = 1$) might be specified as simple as follows.

$$s_{\mathtt{init}} \to \mathtt{mk}_S(0,0,0),$$
$$\delta(\mathtt{mk}_S(q_2,q_1,q_0),\mathtt{mk}_I(\mathtt{pe},i_2,i_1,i_0)) \to \mathtt{mk}_S(d_2,d_1,d_0)$$
where
$$d_2 = \mathtt{pe} * i_2 \oplus \neg\,\mathtt{pe} * \mathtt{sum}(q_2,0,c_2), \qquad c_2 = \mathtt{carry}(q_1,0,c_1),$$
$$d_1 = \mathtt{pe} * i_1 \oplus \neg\,\mathtt{pe} * \mathtt{sum}(q_1,0,c_1), \qquad c_1 = \mathtt{carry}(q_0,0,c_0),$$
$$d_0 = \mathtt{pe} * i_0 \oplus \neg\,\mathtt{pe} * \mathtt{sum}(q_0,0,c_0), \qquad c_0 = 1$$

The where section is shared among all rewrite rules of a module. This is only for notational convenience and gives no further technical advantage. But the mere notational convenience can be enormous: For instance, it turned out quite useful for debugging to have read access functions for every single net in the given netlist, by means of a function symbol $\mathtt{selnet} : \mathbb{N} \times S \times I \to \mathbb{B}$ together with a rewrite rule $\mathtt{selnet}(k, \mathtt{st0}, \mathtt{in0}) \to \mathtt{N}_k$ for the k-th net that refers to the node $\mathtt{N}_k$ in the graph structure of the complete netlist. With sharing of where sections, one needs one copy of the graph structure of the netlist. Without sharing of where sections, one rather needs one copy *per net*.

The reason why we prefer Mealy automata to the simpler Moore automata is the ability to model direct dependency of outputs on inputs. In our case study we encountered such dependencies during decomposition of the netlist into equivalent units.

In practice it turns out very useful to clip the *dependency subgraph* of a netlist. Suppose the netlist is viewed as a graph with nets as nodes and arrows pointing from an input of a box to each output of the box that may depend on the input. By the dependency subgraph we mean the least subgraph that

contains all nodes that occur in the formal specification and that contains for every node in the dependency subgraph, all incoming arrows and their incident nodes; in other words: the nodes that are reversely reachable. Flipflops that are not in the dependency subgraph may therefore be dropped from the state vector. In our case study this meant 419 unidirectional and 52 bi-directional nets and 171 flipflops in the dependency subgraph, as opposed to 3533, 180, and 585, respectively, in the original netlist.

3 How to Separate Propositional Reasoning from Term Graph Rewriting

In this section we prove the simple, but important, theorem that a term graph representing a specification after rewriting represents a propositional formula. That is to say, one may safely delay all propositional reasoning until after term graph rewriting.

A node x in a term graph structure is called *ground* if no variable is reachable from x. It is called a *constructor node* if no node labelled by a non-constructor function symbol is reachable from x. A function symbol $f \in \mathcal{F}\backslash\mathcal{C}$ is called *ground reducible* if every term graph of the form $f(x_1, \ldots, x_n)$ where G where $x_1, \ldots, x_n$ are ground, constructor nodes admits an R-rewrite step.

A term graph rewrite rule, $\ell \to r$, is called *left-linear* if from its left hand side node ℓ each variable is reachable by at most one path.

A basic assumption that we are going to exploit in our theorem is the ability to remove any dependency on propositional values from the rewrite system. Syntactically, we require that every node of type $\mathbb{B}$ at a left hand side of a rewrite rule is a variable (node).

Example 1. For instance, let $f : \mathbb{B} \times \mathbb{N} \to \mathbb{N}$ be a function symbol given by rules $f(0, n) \to g(n)$, $f(1, n) \to h(n)$. Obviously, one cannot rewrite a term $f(t, t')$ at the root unless $t \in \{0, 1\}$. Therefore there is a dependency on the value of the first argument by which dropping propositional reasoning from rewriting leaves a normal form that still contains f symbols.

Every propositional function can be represented by a propositional formula, e.g. in disjunctive normal form. For functions that range in sorts other than $\mathbb{B}$ we propose the following simple fix:

For every sort $s \neq \mathbb{B}$ we assume the existence of a function symbol $\mathtt{switch}_s : \mathbb{B} \times s \times s \to s$. Their defining rewriting rules $\mathtt{switch}_s(0, x, y) \to x$, $\mathtt{switch}_s(1, x, y) \to y$ are suppressed from R because they use $0, 1$ at left hand sides. Instead we assume for all sorts $s, s' \in S, s' \neq \mathbb{B}$ and function symbols $f : \cdots \times s' \times \cdots \to s$ in $\mathcal{F}$ suitable distributivity laws

$$f(\ldots, \mathtt{switch}_{s'}(b, x, y), \ldots) \to b * f(\ldots, y, \ldots) \oplus \neg b * f(\ldots, y, \ldots) \quad \text{if } s = \mathbb{B},$$
$$f(\ldots, \mathtt{switch}_{s'}(b, x, y), \ldots) \to \mathtt{switch}_s(b, f(\ldots, x, \ldots), f(\ldots, y, \ldots)) \text{ else } .$$

Let $\mathcal{D}_{f,i}$ denote the distributive law for $f \in \mathcal{F}$ at argument position i.

Now any set of rules where $0, 1$ occur at left hand sides can be replaced by a set of rules where this is not the case. The dependency on $0, 1$ is encoded in a switch symbol. The purpose of the distributivity laws is to move $\mathtt{switch}_s$ symbols outside from where they block other rewrite steps. On this account a switch function for $\mathbb{B}$ is obviously not needed.

Example 1 (Continued). For instance, let $f : \mathbb{B} \times \mathbb{N} \to \mathbb{N}$ be a function symbol given by rules $f(0, n) \to g(n)$, $f(1, n) \to h(n)$. We replace these rules by $f(b, n) \to \mathtt{switch}_{\mathbb{N}}(b, g(n), h(n))$. Now a term $m + f(b, n)$ rewrites as follows:

$$m + f(b, n) \to m + \mathtt{switch}_{\mathbb{N}}(b, g(n), h(n)) \to \mathtt{switch}_{\mathbb{N}}(b, m + g(n), m + h(n)) \ .$$

We conclude that the use of $0, 1$ or propositional connectives at left hand sides is avoidable. Now we are ready to present our result.

Theorem 1. *Let $(\mathcal{S}, \mathcal{F}, R)$ be a typed term graph rewriting system such that (1) $\mathbb{B} \in \mathcal{S}$; (2) $\mathcal{C}_{\mathbb{B}} = \{0, 1\}$; (3) $PropConn \subseteq \mathcal{F}_{\mathbb{B}}$ is a set of propositional connectives; (4) $Switch = \{\mathtt{switch}_s : \mathbb{B} \times s \times s \to s \mid s \in \mathcal{S}, s \neq \mathbb{B}\} \subseteq \mathcal{F}$ is a set of propositional switches, with distributive laws $\mathcal{D}_{f,i} \in R$ for all $f : s_1 \times \cdots \times s_i \times \cdots \times s_n \to s \in \mathcal{F}$, $s_i \neq \mathbb{B}$; (5) every node of type $\mathbb{B}$ at left hand sides of rules in R is a variable node; (6) R is left-linear; and (7) every function symbol $f \in \mathcal{F} \setminus (\mathcal{C} \cup PropConn \cup Switch)$ is ground reducible. Then every normal form of type $\mathbb{B}$ that contains at most variables of type $\mathbb{B}$ is a term graph that represents a propositional formula, i.e. it only contains reachable nodes that are labelled by propositional variables, propositional connectives, and $0, 1$.*

For instance the carry and the sum bits of a full adder are functions $\mathtt{carry}, \mathtt{sum} : \mathbb{B}^3 \to \mathbb{B}$ specified by the rewrite rules

$$\begin{aligned} \mathtt{carry}(x, y, z) &\to x * y \oplus y * z \oplus z * x, \\ \mathtt{sum}(x, y, z) &\to x \oplus y \oplus z \ . \end{aligned}$$

These rules are of the wanted form, unlike the rules

$$\mathtt{carry}(x, x, z) \to x, \tag{1}$$

$$\mathtt{sum}(0, 0, z) \to z \ . \tag{2}$$

Rule (1) is not left-linear, and Rule (2) violates Condition (5).

We claim that the premises of Theorem 1 are not restrictive. With very little extra effort one can satisfy them. Condition (5) preserves expressive power as we argued above. What is more, propositional variables are only needed left-linearly. Condition (7) means ground reducibility up to propositional connectives.

Proof (Proof of Theorem 1). Let x where G be a term graph of type $\mathbb{B}$ that contains no variable nodes of type $\neq \mathbb{B}$. For convenience assume that all of G is reachable from node x. Now assume that G contains a node x' labelled by a function symbol $f \notin \{0, 1\}$ which is neither a propositional connective nor a propositional variable. Then x where G is reducible by R – so not in normal form – as we are going to show.

First we show that w.l.o.g. $f \notin \mathcal{C} \cup PropConn$. For if $f \in \mathcal{C} \setminus \{0,1\}$ then x' has type $\neq \mathbb{B}$. So G must contain a node labelled by a function symbol that has an argument type $\neq \mathbb{B}$ and result type $\mathbb{B}$. Such symbols are neither in $\mathcal{C}$ by Premise (2) nor in $PropConn$ by definition of propositional connective. So G must have a node labelled by a function symbol $\notin \mathcal{C} \cup PropConn$.

If x' is labelled by a symbol in $Switch$ then one of the distributivity laws applies and we are finished. So suppose that no node is labelled by a symbol in $Switch$.

Now let x' denote a node of minimal height in G, i.e. the length of the shortest path from x to a leaf is minimal, such that x' is labelled by a function symbol $f \notin \mathcal{C} \cup PropConn \cup Switch$. Then there is an assignment $x' = f(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are nodes from which only variable nodes or nodes labelled by a function symbol in $\mathcal{C} \cup PropConn$ is reachable. Let the term graph x' **where** G' be obtained from x' **where** G as follows. Replace every node labelled by a symbol in $\{0,1\} \cup PropConn \cup \mathcal{X}_{\mathbb{B}}$ by a node labelled by 0. Then by Condition (4) x' **where** G' is reducible by R because x' is a ground, constructor node in G'. By Premises (4) and (5) x' **where** G is reducible, too, because the same rule matches it at the same node. Hence the term graph x **where** G is reducible at node x'.

A normal form will be always reached if the term graph rewriting system is terminating, i.e. infinite rewriting sequences $g_1 \to g_2 \to \cdots$ are impossible. It is realistic to expect acyclic term graphs only; then termination can be proven as in term rewriting [Der87]. Compiled acyclic combinatorial hardware is always terminating.

4 FDD evaluation

A requirement is formalized as a term graph of type $\mathbb{B}$ which contains no free variables but of type $\mathbb{B}$. Auxiliary function symbols of any type may be introduced by rewriting rules, and used freely within formal requirements. In a first pass, the requirement is rewritten by the given term graph rewriting rules to a unique *normal form*, i.e. a term graph that cannot be rewritten further.

By Theorem 1, the normal form is a term that consists only of propositional variables and functions. Thus the problem has shrunk to a propositional validity problem, which is solved in the second pass of the procedure. As term rewriting, when applied to propositional connectives, performs much worse than propositional solvers, it is sensible to exclude the respective term rewriting rules from the rewriting process.

Numerous methods are known that solve the propositional validity problem, e.g. the Davis-Putnam procedure (see e.g. [ZBH96]), ordered binary decision diagrams (BDDs) [Bry92], or Stålmarck's method [Sta94,SS90]. We use our implementation of *ordered functional decision diagrams* (FDDs) [KSR92,Keb94].

FDDs differ from BDDs in the following way. A BDD node can be a leaf $\boxed{0}$ or $\boxed{1}$, with the obvious meanings, or an inner node $n = (\ell, x, r)$ with the meaning $[n] = [\ell] * x \oplus [r] * \neg x$. FDD nodes have the same syntax but the meaning

of an inner node is rather $[n] = [\ell] * x \oplus [r]$. The two are mutually obtained by application of the Reed/Muller transform.

The first author has implemented an FDD package in C++, and fit it into a tool for term graph rewriting and inductive reasoning, JRAP. We take it that our choice for FDDs is not critical, i.e. we could have as well used any other complete propositional satisfiability checker.

5 Case Study

We illustrate our method at an industrial case study: a fragment of the IBM S/390 Clock Chip [Spr89, Section 2.8]. The fragment of the chip that we verified is concerned with the collection of set and reset pulses in order to get a synchronized stop signal for all chips of the Capitol chip set: the set of chips that constitute the S/390 microprocessor.

5.1 Informal Description of the Fragment

We are given an informal requirement stating that after a pulse of length one on any of some 22 lines, each of some 30 output lines goes HIGH; after a pulse of length two it goes LOW.

The netlist is given in IBM's netlist language VIM, in technology independent style. *Technology independence* here means that building blocks are broken down to the usual propositional connectives and D-type flipflops, rather than sophisticated blocks that are tailored for efficient layout. It turned out that technology independent style is more amenable to verification since it contains exploitable symmetry from the design phase that adaption to technology may wipe out.

To simplify reasoning we replace the given master/slave clocking scheme by a single clock. We trust that more complex clockings put no insurmountable problem.

Inspection of the chip fragment shows that for each input there is an identical stage that determines whether a pulse of length one or two has been issued at this input. The results are OR-ed for all inputs; they yield set and reset signals, respectively, for a RS type flipflop. Each output of the fragment is preceded by a separate flipflop that buffers the output of the RS type flipflop.

The original VIM netlist text has approximately 25,000 lines or 1 megabyte of ASCII. The specification text for the Mealy automaton encoding the clipped dependency subgraph has 2001 lines or 78 kilobyte.

Intuitively it is fairly clear that the specification should hold. The main problem with the verification is complexity caused by repetition. Let us now briefly discuss how the requirements can be formalized and proved in our framework.

5.2 A First Test

A very first question to ask to the reasoning apparatus may be the propositional function that characterizes the value of some output of the fragment. Such ques-

tions increase faith in the correctness of the implementation and give an estimate of the expected size of the verification problem.

Suppose we might want to know the value of the output no. 22 after two ticks:

$$\mathtt{sel}_O(22, \mathtt{bdo}) \qquad \text{where} \qquad \text{(OUT1)}$$
$$\begin{aligned}
\mathtt{st0} &= \mathtt{mk}_S(\mathtt{N2}, \mathtt{N3}, \ldots, \mathtt{N29}, \mathtt{N50}, \mathtt{N51}, \ldots, \mathtt{N164}, \mathtt{N238}),\\
\mathtt{bdi0} &= \mathtt{mk}_I(\mathtt{B0I0}, \mathtt{B0I1}, \ldots, \mathtt{B0I51}),\\
\mathtt{bdi1} &= \mathtt{mk}_I(\mathtt{B1I0}, \mathtt{B1I1}, \ldots, \mathtt{B1I51}),\\
\mathtt{st1} &= \delta(\mathtt{st0}, \mathtt{bdi0}),\\
\mathtt{bdo} &= \lambda(\mathtt{st1}, \mathtt{bdi1})
\end{aligned}$$

The first line specifies the 22nd bit of the output bit vector **bdo** explained in the where part. The 4th assignment specifies the next state, **st1**, as the result of a state transition given a state **st0** and an input **bdi0**. The line thereafter specifies the output, **bdo**, one transition later. The first three assignments are just there to bind the variables **st0**, **bdi0**, and **bdi1**, so as to ensure that only free variables of type $\mathbb{B}$ remain.

This example also shows how one can speak about time in our framework. The state vector **st0** at time 0 is given as a propositional vector directly. In the same way the input vectors at various time instants are given. The later state vectors are obtained by state transitions, e.g. $\mathtt{st1} = \delta(\mathtt{st0}, \mathtt{bdi0})$, and likewise the output vectors by the output function. With **bdo** denoting the the output vector at time 2, the value of the 22nd output at time 2 is denoted by $\mathtt{sel}_O(22, \mathtt{bdo})$.

The value stored in the k-th D-type flipflop during state **st0** might be denoted by $\mathtt{sel}_S(k, \mathtt{st0})$; its value one step later by $\mathtt{sel}_S(k, \mathtt{st1})$. In order to get acess to still later values one simply adds defining equations for the corresponding states. However there is also a shortcoming with our approach: Values "somewhere" in the future or in the past cannot be expressed; only fixed time intervals are denotable.

The written-out formal requirement specification text of OUT1, including some comments, is approximately 15.8 kilobytes large. JRAP reads the netlist and requirement specifications in 21 seconds, takes 3 term graph rewriting steps – one each to expand δ, λ, and $\mathtt{sel}_O$ – and evaluates the normal form, **B022 where** $\mathtt{B022} = \mathtt{N238} + \neg 1$, to an FDD that represents **N238**. Rewriting and FDD evaluation happen in a fraction of a second if logging is disabled.

5.3 Symmetry of Outputs

A typical symmetry claim states that each of the 30 outputs of the chip carries the same signal. Then by symmetry the remaining propositions need only speak about one such output.

The symmetry claim is conveniently abbreviated by a propositional function $\mathtt{symmout} : S \times I \to \mathbb{B}$ that says whether in the given state with given input every

output yields the same value:

$$\begin{aligned}
&\texttt{symmout}(\texttt{st0}, \texttt{bdi0}) \rightarrow \\
&\quad (\texttt{sel}_O(22, \texttt{bdo}) \Leftrightarrow \texttt{sel}_O(23, \texttt{bdo})) * \\
&\quad (\texttt{sel}_O(22, \texttt{bdo}) \Leftrightarrow \texttt{sel}_O(24, \texttt{bdo})) * \ldots \\
&\quad * (\texttt{sel}_O(22, \texttt{bdo}) \Leftrightarrow \texttt{sel}_O(51, \texttt{bdo})) \qquad \text{where} \\
&\texttt{st0} = \texttt{mk}_S(\texttt{N2}, \texttt{N3}, \ldots, \texttt{N29}, \texttt{N50}, \texttt{N51}, \ldots, \texttt{N164}, \texttt{N238}), \\
&\texttt{bdi0} = \texttt{mk}_I(\texttt{BOI0}, \texttt{BOI1}, \ldots, \texttt{BOI51}), \\
&\texttt{bdo} = \lambda(\texttt{st1}, \texttt{bdi1})
\end{aligned}$$

In fact the proposition $\texttt{symmout}(\texttt{st0}, \texttt{bdi0})$ is true only in states reachable from the initial state. To take this into account we prove the proposition for the initial state and show that its validity for an arbitrary state entails its validity for the next state. Given that P is a proposition on states these can be encoded as

$$\begin{aligned}
&P(s_{\texttt{init}}) * (P(s) \Rightarrow P(\delta(s, i))) \qquad \text{where} \\
&\quad s = \texttt{mk}_S(s_{n-1}, s_{n-2}, \ldots, s_0), \\
&\quad i = \texttt{mk}_I(i_{m-1}, i_{m-2}, \ldots, i_0)
\end{aligned}$$

Observe that by asking for the validity of this formula, the variables s_k and i_k are implicitly universally quantified.

After 2288 steps of rewriting (in 16.30 seconds) and subsequent FDD evaluation (in less than a second) one obtains the expected result $\boxed{1}$. The complete text of `symmout` and the two claims is 955 lines or 50 kilobytes.

5.4 Symmetry of Inputs

It is more difficult to prove symmetry in the inputs since there is only *equivalence* up to renaming of inputs. We pursued the following way. First we specify for each input number $1 \leq k \leq m-1$ a map $\phi_k : S \rightarrow S$ which we intend to describe a bisimulation of the given machine with the machine where the first and the k-th input are exchanged. Intuitively, to each flipflop affected at some moment by the first input we assign the flipflop affected at that moment by the k-th input. Admittedly this requires some inspection and may be tedious. Once ϕ_k is known one may claim bisimulation and equality of outputs:

$$\begin{aligned}
&(s_{\texttt{init}} =_S \phi_k(s_{\texttt{init}})) * (\phi_k(\delta(s, i)) =_S \delta(\phi_k(s), i')) * (\lambda(s, i) =_O \lambda(\phi_k(s), i')) \\
&\qquad\qquad \text{where} \\
&s = \texttt{mk}_S(s_{n-1}, s_{n-2}, \ldots, s_0), \qquad i = \texttt{mk}_I(i_{m-1}, i_{m-2}, \ldots, i_0), \\
&\qquad i' = \texttt{mk}_I(i_{m-1}, \ldots, i_{k+1}, i_0, i_{k-1}, \ldots, i_1, i_k)
\end{aligned}$$

Here bitwise equality of states and of outputs are assumed specified as propositional functions $=_S, =_O$, respectively, in the usual way:

$$\begin{aligned}
&(\texttt{mk}_S(s_{n-1}, s_{n-2}, \ldots, s_0) =_S \texttt{mk}_S(s'_{n-1}, s'_{n-2}, \ldots, s'_0)) \rightarrow \\
&\quad (s_{n-1} \Leftrightarrow s'_{n-1}) * (s_{n-2} \Leftrightarrow s'_{n-2}) * \cdots * (s_0 \Leftrightarrow s'_0)
\end{aligned}$$

The text of specification of ϕ_{10} and the claim for $k = 10$ has a length of 920 lines and a size of 54 kilobytes. JRAP needs 25.7 seconds to parse the text; and a fraction of a second for rewriting (6 steps) and FDD evaluation.

5.5 A Pulse yields a "Set" Signal

Let SET5 be the formalized claim that the sequence 010 (011) on the first input sets the first output. In the same spirit one may design a claim that the sequence 011 on the first input resets the first output.

Reading SET5 takes 24 seconds, rewriting (7 steps) takes a fraction of a second. The claim SET5 is surprisingly hard for the propositional solver. With the two hash tables' size of 3 megawords each, FDD computation fails after having filled the hashtables (in 3 minutes). Note that the number of propositional variables is 480, well above 200 which is commonly taken a realistic upper bound of the number of variables a BDD-based propositional solver can handle.

The claim can nevertheless be solved if one allows rewriting to apply erasing rules like $x * 0 \rightarrow 0$, $x + 1 \rightarrow 1$, or $x \oplus x \rightarrow 0$, as well as an assorted set of auxiliary rules, so as to decrease the effort for the subsequent propositional reasoning. Occasional application of erasing rules tends to simplify the term significantly if the term t instantiated for x is large because an FDD needs no longer be constructed for t. Running JRAP then takes 581 rewrite steps in 55 seconds to produce the trivial normal form 1 that leaves nothing to do for the propositional pass.

The fact that a set (reset) signal is issued *only* by a 010 (011) pulse, is another proof obligation NOSET5. We must skip its detailed presentation for space reasons. Let us just report that the propositional part was particularly difficult, and required a master-tailored variable ordering. Parsing the term (457 lines or 20 kilobytes) by JRAP took 24 seconds, rewriting 281 seconds (1290 steps), and FDD computation (with the right ordering) a fraction of a second.

Conclusion and Related Work

The use of rewriting techniques for hardware verification started with Hsiang's term rewriting system for Boolean rings [Hsi85]. Hsiang observed that the completion procedure of Knuth and Bendix can be turned into a refutational theorem prover. Narendran and Stillman [NS89] and Chandrasekhar, Privitera, and Conradt [CPC87] used this method, enhanced by a few shortcuts, to verify hardware. A survey of this method is given in [HKR92].

Other implemented systems that combine rewriting with built-in propositional reasoning are PVS [ORR+96] and NQTHM [BM84]. Both use BDDs for propositional and temporal reasoning, and term rewriting for inductive theorem proving. The systems have been used successfully to verify data path hardware like divisors and square root algorithms [RSS96,Rus98]. Neither NQTHM nor PVS know term graphs although NQTHM has a LET construct to conveniently address large terms. PVS allows higher-order functions and has a model checker,

NQTHM and JRAP are first-order and have no model checker. PVS and JRAP share a typing discipline.

The work reported here is a development and variation of Bündgen/Küchlin's method to verify the Sparrow processor [BK95,BK96], and its application to industrial hardware. Bündgen and Küchlin translated netlist (BLIF) code into a term rewriting system. This rewriting system, fed into the ReDuX tool, was then the basis to reason about formal specifications.

Bündgen and Küchlin found out that for validity of a term of propositional type the completion procedure is not needed. Provided that the system is confluent and terminating – properties that can easily be deduced in the special case – the pure rewriting to normal form suffices. This is a particularly important observation because it means that the expensive associative-commutative unification algorithm is not needed. Moreover it turned out that dedicated propositional solvers perform substantially better than rewriting by Hsiang's system.

Specification languages that do not support term graphs cannot express something like an assignment $o = f(i_1, i_2, \ldots, i_n)$. In ReDuX rather these were modelled by term rewriting rules $o \to f(i_1, i_2, \ldots, i_n)$ whence nets o had to be modelled by new function symbols. Nets in a sequential hardware were designed as functions from time ($= \mathbb{N}$) to $\mathbb{B} = \{0, 1\}$, with rewrite rules $o(t) \to f(i_1(t), i_2(t), \ldots, i_n(t))$ for combinatorial components and $q(0) \to q_{\mathtt{init}}$ and $q(t+1) \to d(t)$ for D-type flipflops. Structure sharing during rewrite steps and memorization of intermediate results have made the performance good enough to verify the *Sparrow*, a small microprocessor that has actually been built up and tested by students during a practice.

The use of term graph rewriting, rather than term rewriting, and the separation of rewriting and propositional reasoning are the two distinguishing features of our approach as opposed to Bündgen/Küchlin's. The idea to use term graph rewriting and state transition functions originated from experience with Geser's 8085 specification [Ges87]. Term graphs allow for an adequate encoding of hardware, however not as functions from time to $\mathbb{B}$ but rather as explicit state transformers. The meaning of Theorem 1, the separation theorem, is above all the simple, efficient cooperation of rewriting with propositional reasoning. Of course, the propositional solver, built up as an FDD evaluator, may as well be replaced by any other propositional solver.

One further aim in the long run is to prepare inductive reasoning for use in hardware verification. In spite of various efforts inductive reasoning has not yet been successfully used.

References

[BCDM86] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, 35(12):1035—1045, 1986.

[BCL+93] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic Model Checking for Sequential Circuit Verification. Technical Report CMU-CS-93-211, Carnegie Mellon University, July 1993.

[BK95] Reinhard Bündgen and Wolfgang Küchlin. Term rewriting as a tool for hardware and software codesign. In Jerzy Rosenblit and Klaus Buchenrieder, editors, *Codesign — Computer-Aided Software/Hardware Engineering*, pages 19—40. IEEE Press, 1995.

[BK96] Reinhard Bündgen and Wolfgang Küchlin. Verification of the Sparrow processor. In *Proc. Engineering of Computer-Based Systems (ECBS)*. IEEE Press, 1996.

[BM84] Robert S. Boyer and J. Strother Moore. Proof-checking, theorem-proving, and program verification. *Contemporary Mathematics*, 29:119–132, 1984.

[Bry92] R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Comp. Surveys*, 24(3), Sept. 1992.

[Cou94] Bruno Courcelle. Graph rewriting: An algebraic and logic approach. In van Leeuwen [vL94], chapter 5, pages 193–242.

[CPC87] Mandalagiri S. Chandrasekhar, John P. Privitera, and Kenneth W. Conradt. Application of term rewriting techniques to hardware design verification. In *24thACM/IEEE Design Automation Conference (DAC)*, pages 277–283, 1987.

[Der87] Nachum Dershowitz. Termination of rewriting. *J. Symb. Comput.*, 3(1&2):69–115, Feb./April 1987. Corrigendum: 4, 3, Dec. 1987, 409-410.

[DJ94] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In van Leeuwen [vL94], chapter 6, pages 243–320.

[Ges87] Alfons Geser. A specification of the intel 8085 microprocessor — a case study. In Martin Wirsing and Jan A. Bergstra, editors, *Algebraic methods: Theory, Tools, and Applications*, LNCS 394, pages 347–402. Springer, June 1987.

[Gup92] A. Gupta. Formal Hardware Verification: A Survey. *Formal Methods in System Design*, 1992.

[HKR92] Jieh Hsiang, Hélène Kirchner, and Michael Rusinowitch. The term rewriting approach to automated theorem proving. *J. Logic Programming*, 14:71–99, 1992.

[Hsi85] Jieh Hsiang. Refutational theorem proving using term-rewriting systems. *Artificial Intelligence*, 25:255–300, 1985.

[Keb94] Udo Kebschull. *Verhaltensbasierte und spektrale Logiksynthese mehrstufiger Schaltnetze unter Verwendung von Binärbäumen.* PhD thesis, Universität Tübingen, D, June 1994.

[KSR92] Udo Kebschull, Endric Schubert, and Wolfgang Rosenstiel. Multilevel logic synthesis based on functional decision diagrams. In *Proc. European Design Automation Conference (EURO-DAC)*, 1992.

[NS89] Paliath Narendran and Jonathan Stillman. Formal verification of the Sobel image processing chip. In Graham Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 92–127. Springer-Verlag, New York, NY, 1989.

[ORR+96] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. 8th Conf. Computer Aided Veri,cation (CAV)*, LNCS 1102, pages 411–414. Springer, 1996.

[RSS96] H. Rueß, N. Shankar, and M. K. Srivas. Modular verification of SRT division. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 123–134, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[Rus98] D. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level speci,cation of the AMD-K7^{TM} floating-point division and square root instructions. Technical report, 1998. At URL `http://www.onr.com/user/russ/david/k7-div-sqrt.html`.

[Spr89] Wilhelm G. Spruth. *The design of a microprocessor.* Springer, 1989.

[SPv93] M. Ronan Sleep, M. J. Plasmeijer, and Marko C. J. D. van Eekelen, editors. *Term Graph Rewriting: Theory and Practice.* John Wiley & Sons, 1993.

[SS90] Gunnar Stålmarck and M. Säflund. Modelling and verifying systems and software in propositional logic. In B. K. Daniels, editor, *Proc. Safety in Computer Control Systems (SAFECOMP)*, pages 31–36, Gatwick, UK, 1990. Pergamon Press.

[Sta94] Gunnar Stålmarck. System for determining propositional logic theorems by applying values and rules to triplets that are generated from Boolean formula, 1994. United States Patent number 5,276,897; see also Swedish Patent 467 076.

[vL94] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume B (Formal Models and Semantics). Elsevier - The MIT Press, paperback edition, 1994.

[Wir94] Martin Wirsing. Algebraic specification. In van Leeuwen [vL94], chapter 13, pages 675–788.

[ZBH96] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *J. Symbolic Computation*, 21:543–560, 1996.

Automated Test Set Generation for Statecharts

K.Bogdanov[1], M.Holcombe[1], H.Singh[2]

[1] Department of Computer Science
The University of Sheffield
Regent Court, 211 Portobello St.
Sheffield S1 4DP, UK
[2] Daimler-Benz AG, Research and Technology
Alt-Moabit 96a
10559 Berlin, Germany

Abstract. In this paper we present the application of the formal testing method to the statechart notation. The method is proposed for deriving test sets for complex statecharts, i.e. containing hierarchy and concurrency, using a 'divide and conquer' strategy. Initially, test cases are generated for simple statecharts and then these test cases are 'merged' to derive test cases for complex statecharts. They are then populated with test data. Methods for generating test cases for simple statecharts and for 'merging' of such test cases, are described using a simple example. The blackbox test method presented is easy to automate.

1 Introduction

As the performance of programmers increases through the use of more powerful tools and rapid systems prototyping, testing takes increasingly more time. Complete replacement of the phase of testing by formal proofs does not seem to be feasible due to the difficulties in the construction of proofs and a lack of scalable automation. Additionally, the initial requirements may change and the entire proof will have to be redone. The correct operation of the hardware and an operating system under which a system is running may also be questioned and both are hard to account for using complete proofs. Rather often testing is done when the whole system has been coded although the most expensive errors to correct are often introduced early in the lifecycle. Thus a method which allows the designer to automatically test the partially designed and implemented system will be of a particular value.

The test method proposed in the paper addresses these problems and tests that the implementation of a statechart design produces the same output as the design when exposed to the same inputs. By testing we mean applying inputs derived from the design, to an implementation under test and making sure that the output observed is the same as that required by the design. The method uses a theoretically well-founded method of testing X-machines [8].

In Sect. 2 we give an introduction to statecharts [3, 9] using an example of a simple tape recorder. Later, in Sect. 3 we introduce the testing method and

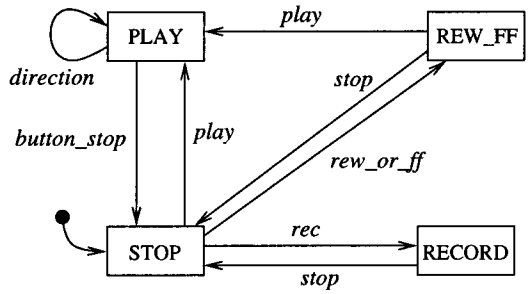


Fig. 1. An example of a tape recorder modelled as a statechart

describe it in some detail in Sect. 4 and Sect. 5. Section 6 concludes the paper and gives some possible directions for future work.

2 Statecharts

Statecharts is a specification language derived from finite-state machines. The language is rather rich in features including state hierarchy and concurrency. Transitions can perform nontrivial computations unlike finite-state machines where they contain at most input/output pairs.

Consider a simple tape recorder model capable of doing all the standard functions like *play*, *rewind*, *fast forward*, *stop* and *record* as well as changing a side of a tape when the button play[1] is pressed during the playback or when a tape ends. A possible statechart is shown in Fig. 1. *STOP* is the initial state. Transition names are selected to reflect user actions, i.e., *play* occurs when the user presses the play button, *rew_or_ff*- either rew or ff buttons. The *direction* transition is triggered by the play button to change the side of a tape or by the tape_end event when the current side has ended.

The tape recorder communicates with a keyboard and a tape drive. It serves as a controller which interprets user's button presses and sends appropriate commands to a tape drive. Input variables are events play, stop, rec, rew, ff and tape_end. Output variables are *ff_direction* and *operation*. They are the commands which tell a tape drive what to do. The *operation* output can have one of the following values: $stop, play, record, move$. The boolean variable *ff_direction* specifies the direction of a tape, with *true* meaning forward tape movement. During playback or recording it also implies the side, with side A being played or recorded forward and B - backward. The communication to a tape drive is bidirectional: the controller is notified when a tape is stopped using the event tape_end.

[1] The names of buttons and events are given in sans-serif font.

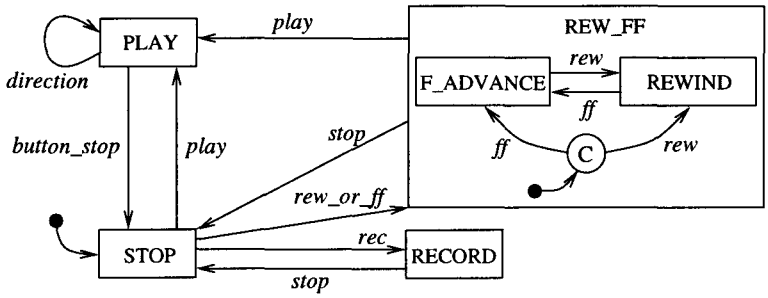


Fig. 2. The tape recorder with a substate statechart

2.1 Transitions

It is possible to specify the *stop* and *button_stop* transitions as

$$stop : \textsf{stop or tape_end}/operation := stop$$
$$button_stop : \textsf{stop}/operation := stop$$

The part before '/' sign means the *trigger*, i.e. the condition which is required to become *true* for a transition to occur. When a transition fires, operation carried out is called an *action*. It is specified after the '/' sign. For the stop transitions above, actions set *operation* to *stop* in order to stop the tape.

2.2 Connectors

Statecharts may contain transitions with much functionality in common. Using **C**, **S** and junction connectors it is possible to separate common parts. In our example the connector is used in state *REW_FF* to enter an appropriate state when requested by a user.

2.3 Hierarchy

An example of state hierarchy is shown in Fig. 2. There is a statechart in the state *REW_FF* which describes a behaviour of the tape recorder when it is in that state. When we enter *REW_FF* by taking the *rew_or_ff* transition, it terminates at the border of *REW_FF* and does not lead to any of the states inside. The blob indicates the beginning of a default transition which is taken in this case. Usually it just points at some state to be entered. Our case is more complicated as the default transition enters a **C** connector such that the state entered depends on the button pressed by a user. A state containing a statechart is referred to as an *OR* state while that without any, a *basic* one.

A statechart within a state (we call it a *substate statechart*) is left when a transition from a state it is in, is taken. For example, if we take the *play*

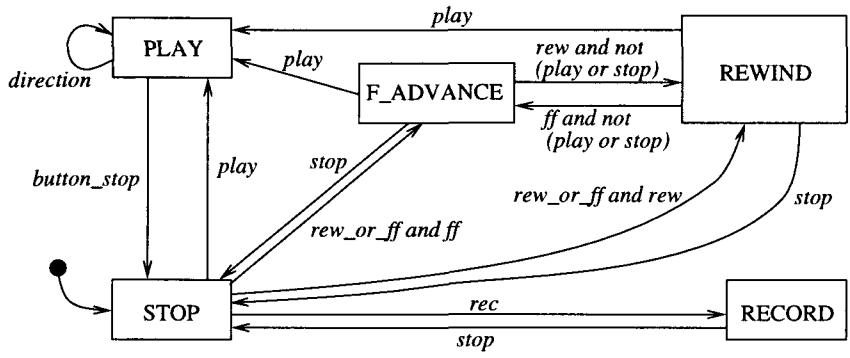


Fig. 3. The tape recorder with the flattened statechart

transition, the *REW_FF* state is left regardless of the state, *F_ADVANCE* or *REWIND*, we were in. The equivalent statechart to that in Fig. 2 is shown in Fig. 3 where its hierarchy and connectors are removed. To do that, the state *REW_FF* has to be replaced by its contents. The outgoing transitions *play* and *stop* and incoming *rew_or_ff* need to be replaced by four corresponding ones. Hierarchy of states imposes priorities on transitions; to retain these priorities, transitions between *F_ADVANCE* and *REWIND* have been appropriately modified in Fig. 3. Furthermore, all transitions of the statechart are free of connectors. Transitions in Fig. 3 represent transitions which are taken during steps in the original statechart in Fig. 2 because, while making a step and taking transitions, we expect to enter a state rather than get stuck in a connector. Such transitions are called *full compound*. In the case where a statechart has no connectors, like the one in Fig. 3, all its transitions are full compound.

2.4 Concurrency

Statecharts have constructs to express concurrency. For example, consider an extension of the tape recorder which allows a search facility. A user can get the tape to advance forward or backward not only when the tape recorder is idle but also when it is playing or recording. The statechart is depicted in Fig. 4. States containing concurrently executing statecharts are called *AND* states.

2.5 Static Reactions

Statecharts are always complete: if there is no transition enabled from a given state, a *static reaction* in that state is executed. A static reaction can be defined for a state using a trigger/action pair similar to that of transitions. The difference between the two is that a static reaction does not leave or enter any state.

If no transition or static reaction is enabled, nothing happens. In this case we can assume that there is an implicit static reaction which does not do anything.

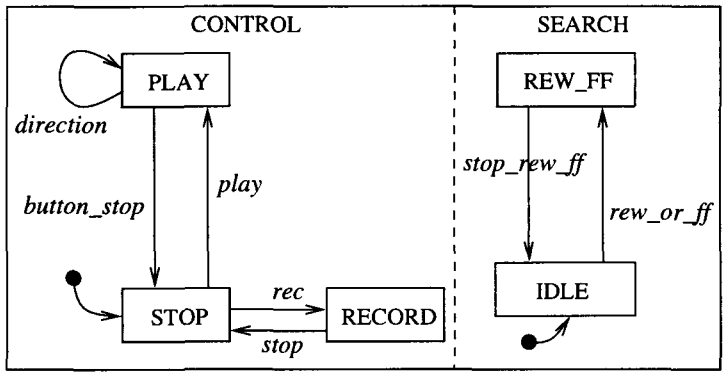


Fig. 4. The tape recorder with the search function

We call such a static reaction a 'do nothing' one. In the case of the tape recorder, 'do nothing' static reactions are in all the states.

3 The Testing Method

3.1 Introduction

The presented testing method for statecharts is based on the X-machine one [8], which has Chow's W method [2] as a foundation. According to the method, we test every transition by visiting every state and generating events to trigger it from that state. Then we should check the state that that transition has led us to, against the designed one. For example, to test the *play* transition from state *REW_FF* to *PLAY*, we should:

- Starting from the initial state *STOP*, enter *REW_FF* by generating rew. The *operation* should change to *move*, assuming that it means a command to a tape drive to begin a tape movement.
- Trigger *play* by generating play and observe the *operation* changing to *play*.
- Test that we entered the *PLAY* state. This can be done by generating tape_end event to trigger *direction* as transition *direction* exists only from the *PLAY* state. After triggering it we should observe the change in *ff_direction* variable.

Note that for every executed transition we need to observe certain output changes which makes us confident that we executed the right transition. For example, if the implementation is faulty as shown in Fig. 5, the *play* transition does not change a state from *REW_FF* to *PLAY*. Thus when trying to trigger *direction* with the tape_end event after invoking *play*, we invoke the *stop* transition. It will produce an output which is different from the output of *direction* (*ff_direction* will not change) and we shall be able to report the fault.

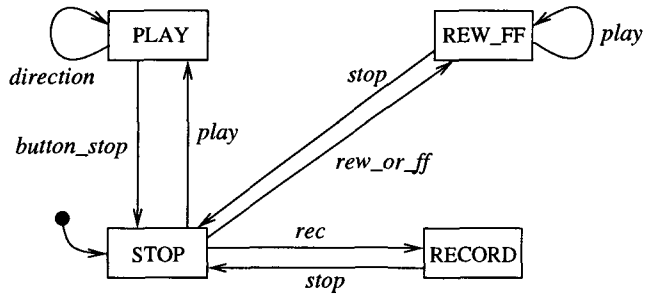


Fig. 5. Faulty implementation of a tape recorder

In addition to testing *play* between those two states, we need to test its existence between *STOP* and *PLAY* as well as to make sure that it does not emanate from any other state. The latter test is needed because in a faulty implementation the transition *play* could exist from some state other than *REW_FF* and *STOP*. To perform this test, we need to visit *PLAY* and *RECORD* states and generate play. This should invoke the *direction* transition in *PLAY* and a 'do nothing' type of static reaction in *RECORD*.

3.2 Design for Test

When applying our test method, we assume that all transitions are implemented correctly. It means that a design may be implemented with a different number of states, transitions could traverse them in possibly random ways, but triggers and actions (Sect. 2.1) of transitions are correctly implemented. Moreover, we assume that we can trigger every transition by supplying an appropriate input (via making changes to externally accessible variables of the system). We also require the combination of a triggering input and an output from a transition to uniquely identify it. This is needed because, having triggered a transition, we need to be sure which one occurred. The requirement of being able to trigger is called *t_completeness* and the one of the input-output pair to identify a transition *output-distinguishability*. When both are satisfied, we can say that the design satisfies the *design for test* condition.

For our example to be *t_complete*, we need an ability to generate tape_end and events generated by buttons. Additionally, having triggered a transition, we should be able to ensure that we did that with the expected one (see the note about *direction*, *play* and Fig. 5 above). In our example this requirement is satisfied, however for more complicated designs we could have to add some inputs and outputs for testing explicitly to satisfy the requirement.

As an example of the triggering problem, consider the transition $TapeCounter = 1035/operation := stop$ which occurs only when an externally inaccessible variable is set to some value. Consequently, we have to artificially

Table 1. The state cover for the tape recorder example

State	Sequence
STOP - the initial state	1 (an empty sequence)
PLAY	*play*
REW_FF	*rew_or_ff*
RECORD	*rec*

augment[2] it by adding an extra input:
trigger or $TapeCounter = 1035/operation := stop$. One might also have to introduce an extra output if the pair **trigger**, $operation = stop$ does not uniquely identify this transition. It is suggested that consideration for satisfying the design for test condition be taken at the design time so that no changes in an implementation are needed in order to do testing.

4 Test Set Generation For Simple Statecharts

4.1 Test Case Generation

Now we shall explain the test case generation for simple statecharts with no concurrency, hierarchy or any connectors like the one depicted in Fig. 1. The result of the test case generation is a set of sequences of transition names such as *rew_or_ff play*. We expect the design for test condition described in Sect. 3.2 to be satisfied. In order to construct a set of test cases, we shall need to build auxiliary sets. There are three of them:

Set of transitions (denoted by Φ) We obviously need to know this. For our example, $\Phi = \{stop, play, rec, rew_or_ff, direction, button_stop\}$.

State cover (denoted by C) To perform testing, we need to visit all states. A *state cover* C is a set of sequences of transition names, such that we can find an element from this set to reach any desired state starting from the initial one. For our example this will be $C = \{1, play, rew_or_ff, rec\}$[3]. In the Table 1 the list of states is shown together with the corresponding elements of C. If the state *RECORD* is reachable only by going through the state *PLAY*, C would be $C_{rp} = \{1, play, rew_or_ff, play\,rec\}$.

Characterisation set (denoted by W) A characterisation set allows us to check the state arrived at when triggering some transition, i.e. if it is the one expected. Above in the description of the testing method (Sect. 3), we had to check if we arrived at the *PLAY* state or not. We did that by trying to follow a *path* (a sequence of transitions) which exists from *PLAY* and not from any other state. For every pair of states, we can construct a path

[2] This word is used to designate a change to the model which does not affect the behaviour restricted to the original sets of inputs and outputs.

[3] We use 1 to denote an empty sequence of transitions.

which exists from one of them and not from the other. For example, for states *STOP* and *REW_FF* we may select *rec* as it exists from *STOP* and not from *REW_FF*. Note that *play* transition exists from both *STOP* and *REW_FF* and thus cannot distinguish them. Such sequences for every pair of states comprise a characterisation set. In our case $W = \{rec, direction, play\}$. Each element of this W is a sequence consisting of a single transition.

With sets C, W and Φ we can construct a set of test cases to be

$$T = C * (\{1\} \cup \Phi \cup \Phi^2 \cup \ldots \cup \Phi^{m-n+1}) * W \tag{1}$$

The meaning of this formula will be explained shortly. In (1), n is the number of states in a design (4 in our example); m is the maximum expected number of states in the implementation, to be described later in this section; set multiplication operation $A * B$ for sets of sequences A, B is defined to be

$$A * B = \{a\,b \mid a \in A, b \in B\} \tag{2}$$

$a\,b$ above means a concatenation of sequences a and b.

If we consider testing a faulty implementation which may have one or more missing states, it is possible to assume that the number of states in such an implementation is at most n. With $m = n$, $T = (C \cup C * \Phi) * W$. This test set deals with visiting every state (the C part of the expression in brackets) and verifying it (by applying W which can distinguish every pair of states). We are then trying every transition from it ($C * \Phi$) and checking the state reached (W).

The way of testing implementations which may contain more states than the appropriate design, $m > n$ is illustrated below. Consider the faulty statechart as shown in Fig. 6. The transition from *ANOTHER_STOP* to *PLAY* is missing. The extra state *ANOTHER_STOP* is reachable only from the *RECORD* one. When testing transitions of the statechart using $(C \cup C * \Phi) * W$, we try them all from all states we can reach using the state cover. Thus we do not try all transitions from *ANOTHER_STOP*. In order to cope with extra states we have to try sequences of more than one transition, such as all pairs of transitions, from every state. From the state *RECORD* one of the pairs of transitions to be tried will be *stop play* which will not bring us to the *PLAY* state as expected. Thus a fault will get revealed. The part of the set of test cases where we try all pairs of transitions can be expressed as $C * \Phi * \Phi * W$. In the case where we assume more than one extra state, we need to make these sequences of transitions longer. For the possibility of $(m - n)$ extra states we arrive at formula (1) for generating sets of test cases.

As stated above, we are trying all possible transitions from every state. The 'do nothing' static reactions (Sect. 2.5) do not correspond to any code and thus their inclusion in the set of test cases is artificial. In our testing method we assume them to be correctly implemented. Static reactions which do have some functionality and consequently are explicitly added to the system design, have to be treated as transitions leaving and entering the states they are defined in.

For testing we assume that the design of the system does not contain redundant states (having the same behaviour as some others), or those with no

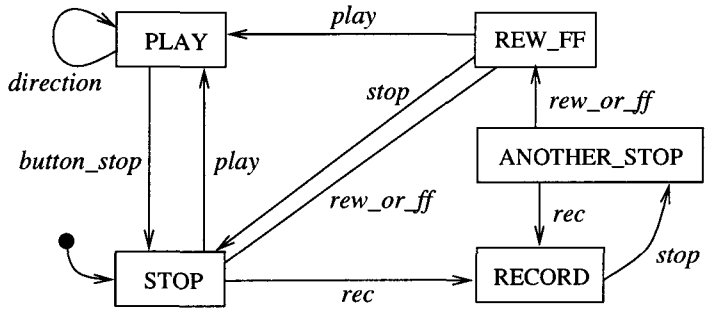


Fig. 6. The faulty implementation with extra states

transitions leading to them. We can call this property *minimality*. The value of m in (1) refers to the maximal number of states in a minimal implementation. We do not require the implementation to possess this property. It is still possible to estimate the number of states in a behaviourally-equivalent one which does satisfy it.

Test case optimisation can be performed employing optimised test case generation methods [1, 5, 6, 10].

4.2 Test Data Generation

Having constructed a set of test cases as described above, we need to generate concrete test data for each of them. The idea is to generate a sequence of test data for each sequence of transitions where each element of this sequence serves as an input for triggering the corresponding transition in the test case under consideration. For a statechart satisfying the design for test condition (Sect. 3.2), the task is relatively easy. Often, we can replace each transition name with an input corresponding to that transition. For example, a possible input sequence corresponding to {*rew_or_ff play*} is {rew play}. We could also use ff instead of rew there. Sometimes, it can be difficult to find an appropriate input sequence for a given sequence of transitions. We can eliminate this problem by augmenting. It is possible to add new triggers to the transitions which could be otherwise difficult to trigger, at the design stage.

To construct expected outputs, we compute the reaction of the design to the input data sequence. For example, a test sequence {rew play} applied in the initial *STOP* state corresponds to *operation* changing to *move* and then to *play*. In the case where no transition or explicit static reaction gets triggered by some input, we expect a 'do nothing' static reaction to occur and the implementation to produce an empty output. Some simple optimisation of the test set can be performed. If we have sequences like rew play and rew play stop, we can eliminate the former as it is the same as the beginning of the latter.

The test data generation method described in this section can be used to generate the test data for the test cases of complex statecharts. The test case generation for them will be described in the next section.

5 Test Case Generation for Complex Statecharts

In the previous chapter we have described testing of simple statecharts. Here we consider more complicated ones. Every feature of statecharts will have a subsection devoted to it.

The most simple approach to testing state hierarchy and concurrency deals with flattening the statechart, i.e. turning it into a behaviourally-equivalent one without substate statecharts and AND states. For example, Fig. 3 depicts a result of flattening of the statechart in Fig. 2. Default connectors are also removed after flattening, leaving a simple (but huge in practice) statechart. Generation of a set of test cases for it is an easy operation as described in Sect. 4.

As an alternative to the above, we propose an approach of incremental test case development. It has the advantage of following the design process and thus providing a possibility of updating the set of test cases to reflect changes made.

5.1 Hierarchy

In order to test state hierarchy, we begin with the generation of a tuple (Φ, C, W) called the *test case basis* for the main statechart and every OR state considering all their substates as basic ones. Afterwards, we combine (we say 'merge') the tuples in a way described below. From the resulting tuple (Φ, C, W) a set of test cases can be constructed following the equation (1) in Sect. 4 or a slightly modified one (3), to follow.

General Approach. We start with the case when no interlevel transitions exist. Statecharts considered may contain default connectors but no others. The tape recorder example is used to illustrate the approach. The state cover set for the statechart in the state *REW_FF*, Fig. 2 can be $C_{REW_FF} = \{1, ff\}$ if we consider the state *REWIND* to be **dflt**. The **dflt** (an abbreviation of 'default') state is the one from where we start traversing states within the *REW_FF* statechart when testing. An alternative state cover set for *REW_FF* is $\{rew, ff\}$ considering the default connector to be a **dflt** state although it can only be treated as an 'instantaneous' state. In our case this selection allows us to construct a smaller set of test cases for the whole system because in order to enter *F_ADVANCE*, we do not have to go through the *REWIND* state. In the following we shall consider the test case basis for *REW_FF* using the revised C_{REW_FF}. Accordingly, the elements of the test case basis for the statechart in the *REW_FF* state are given by

$$\Phi_{REW_FF} = \{ff, rew\}, C_{REW_FF} = \{rew, ff\}, W_{REW_FF} = \{ff\}$$

The elements of the test case basis for the main statechart of the tape recorder (considering the state *REW_FF* to be a basic one) can be seen to be given by:

$$\Phi_{MAIN\,CHART} = \{play, stop, direction, rec, rew_or_ff, button_stop\},$$
$$C_{MAIN\,CHART} = \{1, play, rec, rew_or_ff\}, W_{MAIN\,CHART} = \{rec, direction, play\}.$$

To get the elements of resulting tuple (Φ, C, W), we propose the following 'merging' rules for the above elements:

$$\Phi = FULL_COMPOUND(\Phi_{MAIN\,CHART}) \cup \Phi_{REW_FF},$$
$$C = C_{MAIN\,CHART} \cup \{\text{path to } \textbf{dflt} \text{ state of } REW_FF\} * C_{REW_FF}$$
$$\backslash\{\text{transition from } C_{MAIN\,CHART} \text{ to enter } REW_FF\},$$
$$W = W_{MAIN\,CHART} \cup W_{REW_FF}.$$

A need to trigger all parts of a full compound transition in order to take one is the reason to require all transitions in the resulting test case basis C and Φ be full compound[4] (Sect. 2.3). Consequently, the function $FULL_COMPOUND$ is introduced which makes transitions full compound. This function does its job by adding a possible beginning or a continuation to non-full compound transitions. For *rew_or_ff* in $\Phi_{MAIN\,CHART}$, there are two possible completions as there are two full compound transitions involving *rew_or_ff*,
$FULL_COMPOUND(\{rew_or_ff\}) = \{rew_or_ff\text{-}rew, rew_or_ff\text{-}ff\}$.
'Merging' sets Φ we obtain
$\Phi = \{play, stop, direction, rec, rew_or_ff\text{-}rew, rew_or_ff\text{-}ff, button_stop, ff, rew\}$.
In the set Φ above, transitions which have to be taken in the same step are separated by dashes. The path to **dflt** state of *REW_FF* is *rew_or_ff* and 'merging' the sets for C gives us $C = \{1, play, rec, rew_or_ff\text{-}rew, rew_or_ff\text{-}ff\}$. In order to enter the *REWIND* state from a *STOP* one, we need to trigger both *rew_or_ff* and *rew* transitions. In our example, event **rew** triggers them but in a more complex system more than a single event may have to be generated. The path to the border of *REW_FF* is given by the transition *rew_or_ff* from $C_{MAIN\,CHART}$. The reason for removing it from C is that we have already included paths to enter all states of *REW_FF* in the resulting C. The generation of W involves simply uniting the sets, $W = \{rec, direction, play, ff\}$. As there are no OR states left whose (Φ, C, W) are not incorporated into the resulting tuple, it can be used to generate the set of test cases for the whole system. For example, if the state *F_ADVANCE* had a substate statechart, we would have to 'merge' its test case basis with sets constructed.

It can be observed that the generation process above resulted in the sets possible for the flattened statechart in Fig. 3. In the future the fault detection ability of the sets of test cases obtained via flattening and incremental approach, will be proven.

In the case of interlevel transitions, when constructing sets C and W of the test case basis, we can ignore all interlevel transitions, if possible, but include

[4] W is allowed to contain non-full compound transitions which we expand to full compound during test data generation. The details are omitted from this paper.

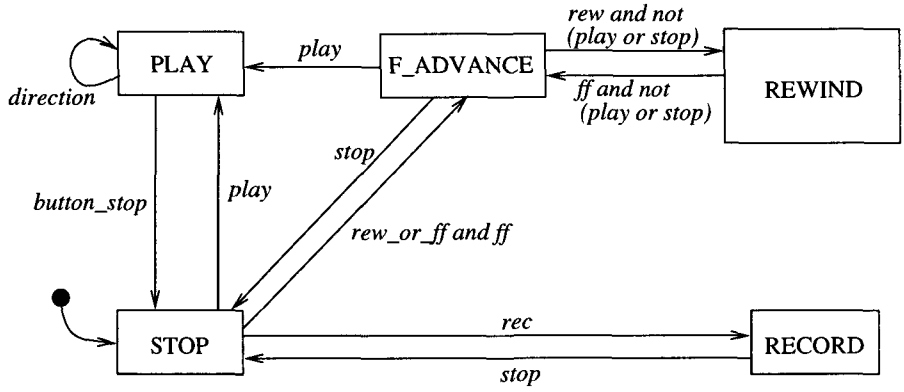


Fig. 7. The faulty implementation which cannot happen if we keep the implementation consistent with every change in the design

them in Φ of the most top-level statechart containing either source or destination of such transitions. The 'merging' procedure is the same as the one for statecharts without interlevel transitions.

Refinement. A design is usually constructed gradually. Initially we may have only a skeleton, then step-by-step we fill it with details. The way we handle consistency between a design and implementation makes a big difference. We could either develop a design in a stepwise manner and having finished it, generate a code, or we could have both design and implementation developed in parallel. The latter means that each change in a design is accompanied by an appropriate change in the implementation of components. In the case of parallel development, we can avoid leaving all testing to the final pre-release stage. Additionally, we can make certain kinds of implementation faults impossible, like the one depicted in Fig. 7. The fault is that transitions *play* and *stop* are not present from the *REWIND* state and *rew_or_ff* cannot enter the *REWIND* state. This fault becomes impossible if initially the design in Fig. 1 was built and implemented and then a statechart was 'placed' in *REW_FF* state (Fig. 2).

With the described restrictions, the test set for such a refinement can be greatly reduced even for our simple example compared to the case when we consider the statechart in the *REW_FF* state to be 'just' a geometrical hierarchy. Faults related to not entering or exiting the state *REW_FF* are not possible; we need to consider only the set of transitions defined inside *REW_FF* when testing it. The set of test cases for development via refinement is given below:

$$T = C_{MAIN\,CHART} * (\{1\} \cup \Phi_{MAIN\,CHART} \cup \ldots \cup \Phi^{m_1-n_1+1}_{MAIN\,CHART}) * W_{MAIN\,CHART} \cup \{rew_or_ff\} * C_{REW_FF} * (\{1\} \cup \Phi_{REW_FF} \cup \ldots \cup \Phi^{m_2-n_2+1}_{REW_FF}) * W_{REW_FF} \quad (3)$$

The first part of the union tests the main statechart treating the state *REW_FF* as a basic one. The second-enters the substate statechart (via *rew_or_ff*) and tests

it separately from the main one. Numbers m_1, n_1 correspond to the expected number of states in the implementation of the main statechart and the number of states in the design of it (4 in our example); m_2, n_2 are the corresponding numbers for the statechart in the *REW_FF* state. The size of the above set of test cases is a half of that computed using formula (1) from the test case basis when we do not consider a hierarchy to be a refinement and do not expect extra states in the implementation. If we do, i.e. all m are greater compared to corresponding n, the difference can be much bigger.

5.2 Concurrency

There are two basic methods of testing concurrent statecharts: via state multiplication and via separate testing of statecharts complemented by the testing of their communication. We consider these approaches in turn.

State multiplication. We begin with the generation of a test case basis for each component of the AND state in Fig. 4. The test case bases for the left concurrent component (*CONTROL*) and the right one (*SEARCH*), follow:

$$\begin{aligned} & \Phi_{CONTROL} = \{direction, stop, button_stop, play, rec\}, \\ C_{CONTROL} = & \{1, play, rec\}, W_{CONTROL} = \{direction, rec\}; \\ & \Phi_{SEARCH} = \{rew_or_ff, stop_rew_ff\}, \\ C_{SEARCH} = & \{1, rew_or_ff\}, W_{SEARCH} = \{stop_rew_ff\}. \end{aligned}$$

To build a test case basis for the whole statechart, we need to 'merge' those of individual components. Sets C and Φ have to be multiplied; sets for W simply get united,

$$\begin{aligned} \Phi = & (\{1\} \cup \Phi_{CONTROL}) \underline{*} (\{1\} \cup \Phi_{SEARCH}) \setminus \{1\} = \{direction, stop, \\ & button_stop, play, rec, rew_or_ff, stop_rew_ff, direction\text{-}rew_or_ff, \\ & stop\text{-}rew_or_ff, button_stop\text{-}rew_or_ff, play\text{-}rew_or_ff, rec\text{-}rew_or_ff, \\ & direction\text{-}stop_rew_ff, play\text{-}stop_rew_ff, stop\text{-}stop_rew_ff, \\ & button_stop\text{-}stop_rew_ff, rec\text{-}stop_rew_ff\}, \\ C = & C_{CONTROL} \underline{*} C_{SEARCH} = \\ & \{1, play, rec, rew_or_ff, play\text{-}rew_or_ff, rec\text{-}rew_or_ff\}, \\ W = & W_{CONTROL} \cup W_{SEARCH} = \{direction, rec, stop_rew_ff\}. \end{aligned}$$

Above, multiplication means concurrent invocation rather than sequential one (2) such that $\{direction\} \underline{*} \{rew_or_ff\} = \{direction\text{-}rew_or_ff\}$. One can observe that the sets constructed are appropriate for the statechart in Fig. 8 which is the equivalent of Fig. 4 without concurrency.

The development process where we design and implement concurrent states and then place the appropriate statecharts in them (refinement), allows us to eliminate the faults where the state entered by some transition is different if we

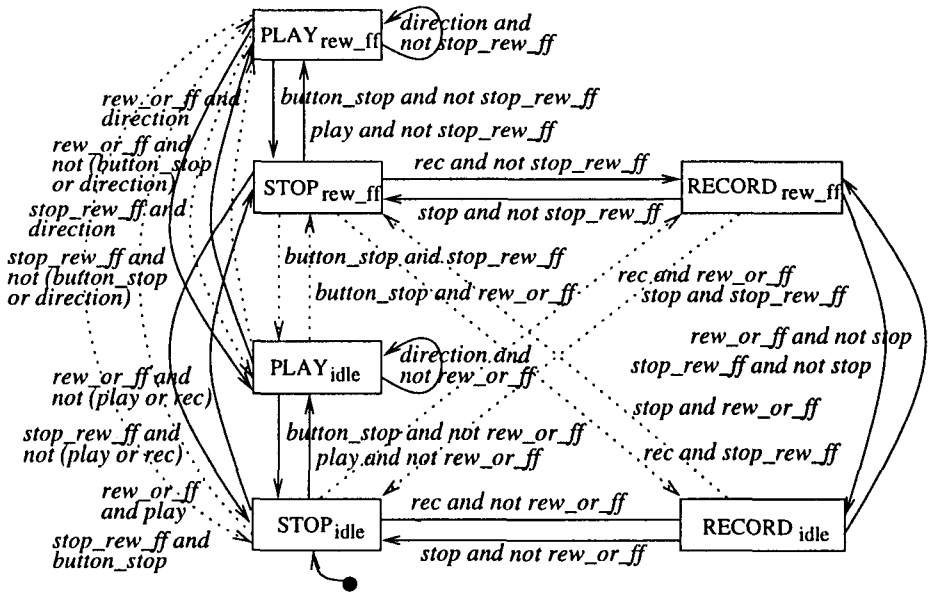


Fig. 8. Testing AND-states via state multiplication

take a transition in the concurrent component in the same step. For example, such a fault could involve transition *rec* erroneously entering the *PLAY* state when executed at the same time as *rew_or_ff*, and the correct state *RECORD* otherwise. For the case of refinement, we test *rec* and *rew_or_ff* but not both of them at the same time. As shown, it is possible to use $\Phi = \Phi_{CONTROL} \cup \Phi_{SEARCH}$ which reduces the size of the set of test cases twice for our example. The transitions on the flattened statechart which we thus do not take, are given in dotted lines in Fig. 8.

Transitions entering or exiting concurrent states can be tested as ordinary interlevel transitions.

Separate testing. In the case of separate testing, concurrent statecharts can be tested separately and process algebraic methods [4] or finite-state machine based methods [7] be applied to test their communication. If the concurrent statecharts exhibit little communication, the set of the test cases is essentially a union of that for the individual concurrent statecharts.

6 Conclusion and Future Work

Although we described the method rather informally, it is based on the X-machine test method [8] which is formal. With some restrictions on the class

of statecharts we deal with, it is possible to make the method guarantee that all faults in the implementation of a statechart design are detected, provided the maximal number of states in the minimal implementation is bounded by some value we use to construct a test set. The additional requirements for the test method are the deterministic design and the design for test condition together with correct implementation of the behaviour of transitions. Also, if the design is nondeterministic, we could augment transitions with test inputs such that it will behave deterministically under the test and thus still perform the testing.

Proofs for the 'merging' rules and reference tool support in JAVA are under construction. The proposed implementation is supposed to verify the necessary conditions and generate a test set for a given statechart. At present the tool developed is limited to generation of a test case basis and a test set for simple statecharts satisfying the design for test condition, like the one in Fig. 3. To show the anticipated scalability of the approach, it will be applied to a much bigger case study of the hifi stereo than the one described here and results made available in a future report.

Acknowledgements

This work is funded by the Daimler-Benz AG, Research and Technology, Berlin. The authors would like to thank Sadegh Sadeghipour for helpful discussions.

References

[1] P. Bernhard. A reduced test suite for protocol conformance testing. ACM Trans. on Software Engineering and Methodology. **3(3)** (1994) 201–220

[2] T. Chow. Testing software design modeled by finite-state machines. IEEE Transactions on Software Engineering. **SE-4(3)** (1978) 178–187

[3] D. Harel. Statecharts: a visual formalism for complex systems. Science of Computer Programming. **8** (1987) 231–274.

[4] M. Hennessy. Algebraic Theory of Processes. MIT Press. (1988)

[5] R. Hierons. Extending test sequence overlap by invertibility. COMPJ: The Computer Journal. **39(4)** (1996) 325–330

[6] R. Hierons. Testing from a finite state machine: extending invertibility to sequences. COMPJ: The computer journal. **40(4)** (1997)

[7] R. Hierons. Testing from semi-independent communicating finite state machines with a slow environment. IEE Proceedings on Software Engineering. **144(5–6)** (1997) 291–295

[8] F. Ipate and M. Holcombe. An integration testing method that is proved to find all faults. International Journal on Computer Mathematics. **63** (1997) 159–178

[9] A. Naamad and D. Harel. The statemate semantics of statecharts. Technical Report. `http://www.wisdom.weizmann.ac.il/Papers/CSreports/reps95/95-31.ps.Z` (1996)

[10] T. Ramalingam, A. Das, and K. Thulasiraman. On testing and diagnosis of communication protocols based on the fsm model. Computer communications. **18(5)** (1995) 329–337

Rigorous Compiler Implementation Correctness: How to Prove the Real Thing Correct

Wolfgang Goerigk and Ulrich Hoffmann

Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Preußerstraße 1-9, D-24105 Kiel, Germany. {wg|uho}@informatik.uni-kiel.de

Abstract. We give a comprehensive technical overview of our work on rigorous verification of compiling specification and compiler implementation of an initial correct binary compiler executable. We will concentrate on implementation verification. Machine program correctness is proved by a special bootstrapping technique with à posteriori code inspection. Our contribution is to perform this work for compilers and, hence, to relieve the application programmer's burden to prove implementation correctness again and again, as this is done today for safety and security critical applications. Once our work has been finished conscientiously and is accepted to reach sufficient mathematical certainty, compilers may be used for proved correct program implementation, safely in the sense that every result a target program execution returns is guaranteed to be correct with respect to the source program semantics.

1 Introduction and Motivation

The major goal of the *Verifix*[1] project on *Correct Compilers* [7] is to develop methods for correct realistic compiler construction for practically relevant source languages and concrete target machines, and to completely verify them down to their binary machine code implementations. The use of computer based systems in safety critical applications justifies and requires the verification of software components. Correct program execution, however, crucially depends on the correctness of the binary machine code executable, and, therefore, on the correctness of compiler programs. This is true for security as well.

In 1984, Ken Thompson, the inventor of Unix, devoted his Turing Award Lecture [27] to security problems due to Trojan horses intruded by compilers and compiler implementations. He shows in detail how to completely hide a small piece of virus code, a Trojan horse, in the binary implementation of a concrete C compiler, not visible in the compiler source code, but reproducing itself when this source code is recompiled in a bootstrapping process. The virus then generates a back door into the executable for the Unix login command, allowing him to use a special password for hacking into the system.

[1] Verifix is a German joint project on *Correct Compilers*. Three research groups at the universities of Karlsruhe, Ulm and Kiel cooperate, funded by the DFG (Deutsche Forschungsgemeinschaft).

No source code inspection, no source code verification will ever find such an intentional error, and it is unlikely that compiler validation can help. Ken Thompson closed his lecture:

> You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code.

In 1988, J Moore [21] pointed out, that full compiler verification has to verify the compiler implementation as well. Unfortunately, the literature gives no sufficient solution so far. The purpose of this paper is to show that rigorous compiler verification down to the binary machine code executable of the compiler is possible and feasible. In order to guarantee full compiler correctness, we have to carefully verify both, the compiling specification and the compiler implementation, and obviously, source code verification of the high level compiler implementation is not sufficient. Thus, for a full compiler correctness proof, we proceed in four steps corresponding to a simple cascaded software engineering model:

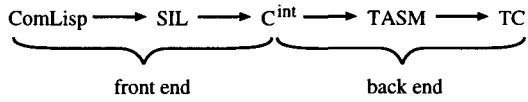
1. define an appropriate notion of *correct compilation* for sequential imperative languages, which guarantees sufficient correctness properties for the target program on concrete target processors with finite resource limitations (preservation of partial correctness or L-simulation, cf. section 2),
2. define a compiling specification $\mathcal{C}$ relating source to target programs, and prove semantically, that $\mathcal{C}$ preserves partial correctness (compiling verification, cf. section 3),
3. construct a compiler program $\pi_{\mathcal{C}}$ in the source language and prove, that $\pi_{\mathcal{C}}$ is a *refinement* (correct implementation) of $\mathcal{C}$ in the sense of preserving partial correctness (correct compiler construction, cf. section 4), and finally,
4. use an existing (unverified) implementation of the source language to execute $\pi_{\mathcal{C}}$, apply $\pi_{\mathcal{C}}$ to itself and bootstrap a compiler executable $m_{\mathcal{C}}$. Check syntactically, that $m_{\mathcal{C}}$ actually has been generated according to $\mathcal{C}$ (compiler implementation verification, cf. section 5).

We have to be very careful in the last step. The correctness of $m_{\mathcal{C}}$ depends on an unsafe compiler execution. Therefore, we additionally check the target program $m_{\mathcal{C}}$ to actually be generated according to $\mathcal{C}$ (à posteriori double checking the results). Correctness of $\mathcal{C}$ and $\pi_{\mathcal{C}}$, and checking $m_{\mathcal{C}}$ guarantee, that the compiler executable $m_{\mathcal{C}}$ preserves partial correctness.

In particular, binary code inspection guarantees the absence of Trojan horses as follows: the virus could either be part of or generated by $m_{\mathcal{C}}$, or both, like in Ken Thompson's scenario. In step 4. above we checked $m_{\mathcal{C}}$ not to contain it. $\mathcal{C}$ and $\pi_{\mathcal{C}}$ are both proved correct (steps 2. and 3.), and $m_{\mathcal{C}}$ is proved to be generated from $\pi_{\mathcal{C}}$ according to $\mathcal{C}$ (step 4.). Thus, $m_{\mathcal{C}}$ neither generates nor contains the virus.

Our paper concentrates on the implementation verification for an initial fully verified compiler implementation from a subset ComLisp [9,6] of Common Lisp into the binary machine code of the Inmos Transputer [15]. Concrete compiling specifications are not defined here; we refer to [10]. Compilation, however, is

described in some detail in section 5. Compiling specification and compiler implementation are both modularized into four compiler phases using three different intermediate languages:



The front end is completely machine independent. It deals with languages without any resource limitations. The first phase generates abstract stack intermediate code, and the second phase translates into a simple imperative language with linear memory. The two back end phases are machine dependent, compiling into concrete assembly code and, finally, generating binary machine code. It is important, that every compilation phase retains the modularization of the code into subroutines.

2 A Precise Description of the Problem

Our implementation and compiler correctness notion is *preservation of partial correctness* [7, 22, 11]. It is adequate and essential for rigorous compiler verification and compiler implementation verification for sequential imperative languages, guided by the intuition, that program execution on real hardware must not deceive the user with respect to correctness of results. Preservation of par-

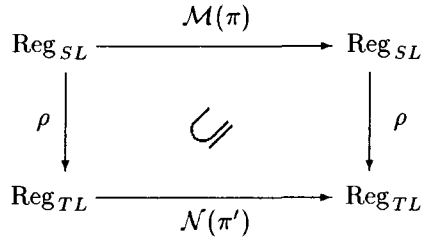


Fig. 1. Compiling Correctness. $\pi \in$ SL and $\pi' \in$ TL denote (imperative) source resp. target programs with semantics $\mathcal{M}(\pi)$ and $\mathcal{N}(\pi')$, relations on the domains of regular state or data spaces $\mathrm{Reg}_{SL} \subseteq Q_{SL}$ and $\mathrm{Reg}_{TL} \subseteq Q_{TL}$. $\rho \subseteq \mathrm{Reg}_{SL} \times \mathrm{Reg}_{TL}$ is a data representation function or relation

tial correctness allows the target machine program to fail with e.g. memory or arithmetic overflow [26]. Whenever a target program execution gives a regular result (does not abort), however, it is guaranteed to be correct. Figure 1 shows the general setting. Let $\cdot;\cdot$ denote relational composition. Then we define π' to be a *correct implementation* of π, if and only if

$$\rho\,;\,\mathcal{N}(\pi') \subseteq \mathcal{M}(\pi)\,;\,\rho\ . \tag{1}$$

This captures the following intuitive requirements:

1. If a target program execution, i.e. π', ends in a regular final state without resource violation, then the source program π can end in a corresponding final state. Any regular target program result is correct w.r.t. the source program.
2. If the source program π can only end in error states, then the target program π' must not end in a regular final state.
3. If the source program π diverges, i.e. cannot terminate, then the target program π' must not end in a regular final state. In the deterministic case this means, that divergent source programs may only be compiled into target programs which either are divergent or end in irregular resource errors.

Since we do not want to rule out concrete finite target hardware processors, we have to admit that target program execution always may end in a resource violation. Partial source program correctness is preserved, whereas total correctness is not in general. Our definition is equivalent to the more elementary requirement

$$\begin{aligned} &\forall\, i_1 \in \mathrm{Reg}_{SL}, i_2 \in \mathrm{Reg}_{TL}, f_2 \in \mathrm{Reg}_{TL} : \\ &\quad (i_1, i_2) \in \rho \ \wedge\ (i_2, f_2) \in \mathcal{N}(\pi') \\ &\quad\Longrightarrow\quad \exists\, f_1 \in \mathrm{Reg}_{SL} \ :\ \ (f_1, f_2) \in \rho \ \wedge\ (i_1, f_1) \in \mathcal{M}(\pi)\ . \end{aligned} \tag{2}$$

A compiling specification $\mathcal{C} \subseteq SL \times TL$ is said to be *correct* in the sense of preservation of partial correctness (*preserves partial correctness*), if, for every $(\pi, \pi') \in \mathcal{C}$, π' is a correct implementation of π.

For our purposes, in particular for *vertical* compiler decomposition into subsequent compilation phases, it is crucial that this implementation notion guarantees compositionality. The little calculation

$$\rho_1; \rho_2; \mathcal{M}(\pi'') \subseteq \rho_1; \mathcal{M}(\pi'); \rho_2 \subseteq \mathcal{M}(\pi); \rho_1; \rho_2$$

for instance shows vertical compositionality (π_1 is compiled to π_1' and further on to π_1'' with representation relations ρ_1 und ρ_2). A similar proof shows horizontal compositionality, which allows for the sequential composition of correct implementations of program parts. Compositionality is a very important property of our modular approach: The correctness results which have to be proved in parts of the proof do not depend on the particular place where they are used.

3 Compiling Verification

Compiling verification establishes the fact that a compiling specification $\mathcal{C}$ is semantically correct, that it preserves partial correctness. Compositionality ease this proof because it can be modularized into different compilation steps and into separate compilation theorems for different language constructs. This fact coincides with experiences from practical compiler construction and with necessities which show up if we trustworthily want to prove compiler implementation

correctness (cf. section 5). It is necessary to keep source and target programs close in order to be able to double check compilation results.

The compiling correctness of both back end phases is proved using the algebraic refinement techniques in [23]. Formalization and mechanical proof assistance including the necessary fixed point theory for the proof sketched in [11] is available and has been documented in [25]. Denotational semantics for ComLisp and SIL is defined in [6]. A simple yet efficient garbage collection scheme (stop-and-copy) has already been implemented and verified [20].

We do not want to go into detail on compiling verification here. Let us from now on assume that we have a proved correct compiling specification $\mathcal{C} \subseteq SL \times TL$, an inductively defined relation between our source and target languages.

4 Compiler Implementation Verification

At this point, $\mathcal{C}$ is far away from an implementation as a binary machine program executable. In the following we will, step by step, construct or generate more concrete correct implementations, decomposing the full implementation problem into small steps. Ideally, every single step handles exactly one crucial implementation decision and, hence, can easily be verified and/or checked, keeping source and target code close.

The crucial idea is, that L-simulation (preservation of partial correctness) is the notion of correct implementation in every single refinement step, for the compiler construction, i.e. the manual implementation of $\mathcal{C}$ as a high level recursive program π_1, but also, and this is essential, for those steps in which we actually mechanically generate intermediate programs using a compiler which preserves partial correctness. The vertical composition theorem (cf. section 1) allows for stacking of diagrams without any further complication, and, hence, we are free to choose an appropriate decomposition.

However, we shall reach an intermediate layer where further manual refinement will become inadequate, where we should use a compiler to generate further low level implementations. At this point, we are suddenly left alone with our rigorous correctness requirement, because there is no fully verified compiler available yet. Fortunately, we are just on the way to produce one which, again fortunately, indeed preserves partial correctness. The question arises how we can make use of our own compiler to help us to mechanically generate low level implementations correctly, although we have not yet finished the correctness proof.

We will show, that this is possible, if we observe some requirements from the very beginning: We have to use an existing programming language, for instance ComLisp, both as source and implementation language, because we want to run the compiler once applied to itself. It has to preserve partial correctness and we must be able to check the results of this one compiler execution, because at this moment our compiler is not yet guaranteed to be correct, although we are quite sure that it will be. Indeed, we are just proving it.

In the following we will go into further detail and show how this procedure actually is worked out for our existing ComLisp compiler.

4.1 Compiler Implementation Correctness

Let m_C be an implementation of C on a compiler host machine HM, written in machine language HL. Then m_C is said to be a *correct implementation* of C, if the diagram in figure 2 commutes in the sense of L-simulation or preservation of partial correctness:

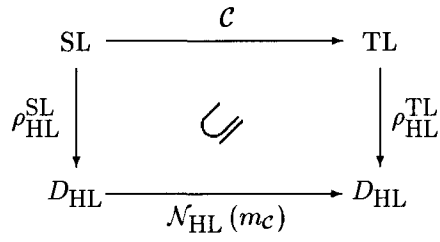


Fig. 2. Compiler Implementation Correctness. $\mathcal{N}_{HL}(m_C)$ denotes the semantics of $m_C \in$ HL. ρ^{SL}_{HL}, ρ^{TL}_{HL} are data representations for proper source and target programs into the data language D_{HL} of the compiler host machine

Let us now specialize this general situation and identify source and implementation language as well as compiler target and host machine language. Since we prove correct compiler construction in high level implementation language and correctness of the binary compiler implementation separately, we can decompose diagram 2 into two steps:

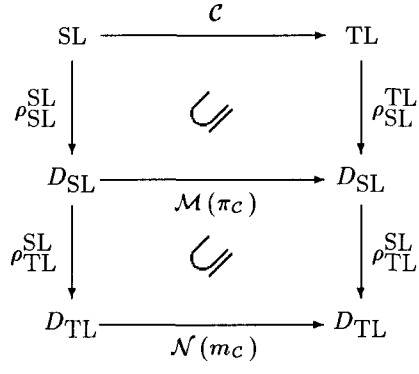


Fig. 3. Implementation Correctness by Vertical Composition. D_{SL} and D_{TL} are the data domains of source and target languages, ρ^{SL}_{SL}, ρ^{SL}_{TL}, and ρ^{TL}_{SL} are the corresponding data representations. Data representation is quite easy if we use Lisp s-expressions

The upper part of diagram 3 corresponds to the correct compiler construction as high level (recursive) program π_C constructed in ComLisp = SL. The lower part corresponds to the compilation of π_C into m_C using our own compiler. Note, that we use two completely different approaches in order to establish L-simulation or preservation of partial correctness in the two parts of the diagram.

4.2 Correct Compiler Construction

Since ComLisp programs are systems of mutually recursive first order function and procedure definitions with strict (call-by-value) parameter passing and abstract s-expression data, we can correctly construct π_C easily [14] by referring to theorems about data refinement and standard program transformations from literature [16, 2]. One of the important future works in the *Verifix* project will be to formalize and to mechanically support compiler construction and the verification of this step. Formal compiler construction will end in a system of recursively defined function definitions, already very close to a ComLisp program.

4.3 Bootstrapping

Since we use an existing language, a proper subset of Common Lisp, as source and implementation language, we can now generate a first (binary) compiler implementation m_C; This process is shown in figure 4 using McKeeman's T-diagrams, specializing TL to the Transputer machine code TC:

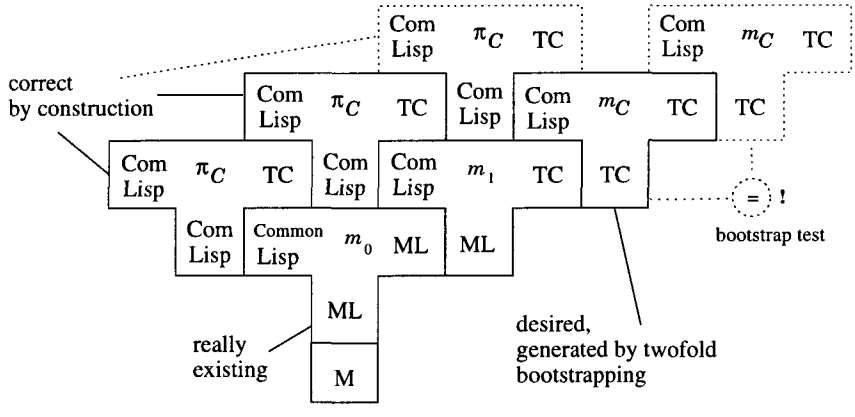


Fig. 4. Bootstrapping the Correct Compiler Implementation. We implement π_C, using an existing Common Lisp compiler m_0, on host machine M, and apply the resulting host machine executable m_1 to π_C in order to generate m_C

But is the compiler implementation m_C correct? A first answer to this question is given by the following *Double Bootstrapping Theorem* [19, 5], which, unfortunately, does not apply in general, because existing optimizing compilers (m_0)

are not verified and in fact often do not preserve partial correctness; they omit e.g. range checks and, hence, the results of a target program execution are unpredictable (chaotic) in error situations. No mathematically precise proposition is known about m_0. However, once we succeeded in implementing an initial fully correct compiler executable preserving partial correctness, this theorem applies for further correct compiler or system software development. Note that termination below means regular termination without runtime error.

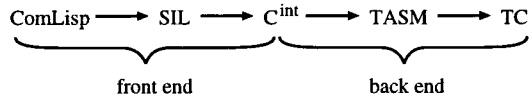
Theorem 1. *If m_0 and π_C both preserve partial correctness, if m_0, applied to π_C, terminates with result m_1, if the host machine executable m_1, applied to π_C, terminates with result m_C, and if the underlying hardware worked correctly, then the compiler implementation m_C preserves partial correctness.*

It is also well known, that a successful bootstrap test is not sufficient to prove the correctness of m_C nor that of m_1 or m_0. Even official validation does not guarantee correctness. The situation is even worse, if we recall Ken Thompson's scenario from the introduction. Thus, something has to be checked manually, since we "can't trust code, that we did not totally create ourselves".

A correct way out uses mathematical à posteriori checking, that m_C has been generated from π_C according to the verified compiling specification C. In particular, this proof by double checking the results will detect any intruded virus. This demonstrates how important full compiler implementation verification down to binary machine code is. We will give a closer look into this proof in the following section. Double checking is a formal technique and compares to manual mathematical proof checking.

5 A Closer Look into Double Checking the Results

In the following we refer to our concrete compiler implementation from SL= ComLisp to binary Transputer machine code TC. Let us recall the structure of this compiler: It proceeds in four separate phases. Each phase is correctly implemented in ComLisp and generates an external syntactical Lisp s-expression representation of the intermediate and target programs.



The first phase transforms procedures and functions with parameters into parameterless procedures and stack technique. The second phase refines data and operations for dynamic Lisp data to a linear memory representation. The third phase resolves control structures and generates linear assembly code. The last phase translates abstract assembly code into concrete machine code. Finally, a boot loader machine program [10] is used to load this machine code representation into the Transputer memory and to execute it.

According to section 4.3 we now use a Common Lisp system to execute the compiler, and to apply it to itself. Hopefully, this execution returns a correct Transputer implementation of the compiler source program. But this is not

guaranteed. Therefore, we now have to check the resulting target code. Because it should be generated according to our own compiling specification $\mathcal{C}$, this is quite easy, because we know exactly, how it should look like. If our check succeeds, we can guarantee full correctness of the compiler executable which we just generated. However, the large step of compiling Lisp programs directly into binary Transputer machine code is actually too large. We would have to check, that the code for e.g. a function definition like

```
(defun f (x y)
  (+ (* x y) 3))
```

which compiles into the binary Transputer machine code

```
75 E0 73 75 E1 73 FA D3 75 52 D5 75 74 F9 A2 21 F0 73
58 71 F9 A2 21 F0 73 30 73 E4 73 31 73 E5 73 32 73 E6
73 33 73 E7 44 70 34 F6 43 73 E6 43 73 E7 44 70 35 F6
73 34 73 E0 73 35 73 E1 75 60 5E D5 75 31 D3 75 30 F6
```

is the correct one. Without any further structure of the target code we would not be able to do this trustworthily. Note, however, that even this check would be purely syntactical. We *need not* analyze the target code semantically. The semantical correctness follows from compiling verification and correct compiler construction together with the successful syntactical code inspection.

Fortunately, the vertical modularization into intermediate languages gives the necessary structure. We will show this using the concrete output of our compiler implementation for the above function, which as an example is actually not part of the compiler. It has no control structure (no loops or conditionals), which slightly simplifies the presentation here because we will not have to check relative jump distances for the assembler code.

5.1 Checking the Front End

The first two compilation steps are machine independent. We start with our original ComLisp function:

```
(defun f (x y)                                          ComLisp
  (+ (* x y) 3))
```

Compiling ComLisp to SIL essentially is the transformation of expressions into postfix form. Our function body will be compiled to `(x y * 3 +)`, augmented with relative positions of variables and intermediate results. The last statement copies the result to the stack position 0. In order to make global checks unnecessary when double checking lower level translations, we already allocate absolute and relative stack memory addresses. Compiling verification guarantees, that the stack principle is a correct implementation of expression evaluation; we need not check this here.

```
(DEFUN F ((X 0) (Y 1))                                          SIL
  ((X 0) 2)
  ((Y 1) 3)
  (* 2 2)
  ('3 3)
  (+ 2 2)
  (%POP 3 0))
```

The next step is data refinement of dynamically typed Lisp data to a linear memory architecture. Relative addresses are multiplied by 2 (tag and value field) and copied pairwise. We have to focus on single SIL statements and compare them with pairs of target statements: In order to copy the content of x from relative position 0 to 2, the target code has to copy the tag field from 0 to 4 and the value field from 1 to 5. Operator calls now become subroutine calls into the runtime system – compiling verification will show that the runtime system procedures are correct operation refinements of the SIL-operators.

```
(DEFUN F (8)                                                    C_Int
  (%%SETLOCAL (%%LOCAL 0) 4)
  (%%SETLOCAL (%%LOCAL 1) 5)
  (%%SETLOCAL (%%LOCAL 2) 6)
  (%%SETLOCAL (%%LOCAL 3) 7)
  (* 4)
  (%%SETLOCAL 3 6)
  (%%SETLOCAL 3 7)
  (+ 4)
  (%%SETLOCAL (%%LOCAL 4) 0)
  (%%SETLOCAL (%%LOCAL 5) 1))
```

This completes the checking for the machine independent front end phases. The next two steps are machine dependent, and we will only show the first step generating Transputer assembly code. The final step generates the machine code above, and it is obvious how to check this.

5.2 Checking the Back End

The first back end phase transforms control structure into linear assembler code with relative jumps. The assembler subroutine body consists of procedure entry code, the main part and procedure exit code, three parts which are structured into three lists in the TASM-code. Entry and exit code share the same pattern for every procedure. This phase also handles resource restrictions of the concrete 32-bit machine. But this is a semantical issue not to be checked here.

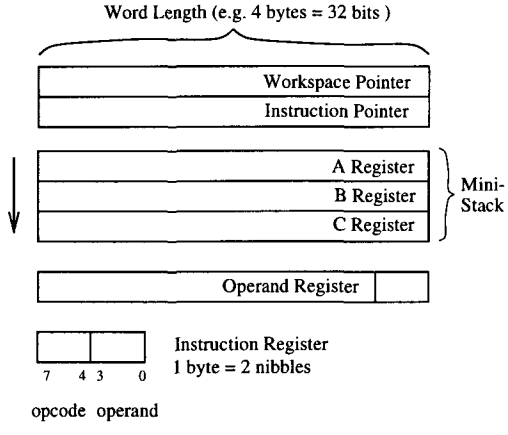
In order to check the main part, we have to compare single C^{int} instructions with small groups of four or five assembler instructions. For instance, the instruction `(%%SETLOCAL (%%LOCAL 0) 4)` (the second line above) is compiled into the instruction sequence `(LDL 3 LDNL 0 LDL 3 STNL 4)` (first line of the main part below), which first loads the *frame pointer*, then the content of relative position 0, which after loading the frame pointer again is finally stored into relative position 4.

```
(%%%DEFCODE F 6                                                    TASM
  (LDL 5 STNL 0 LDL 3 LDL 5 STNL 1 LDL 3 OPR 10 STL 3 LDL 5
   LDNLP 2 STL 5 LDL 5 LDL 4 OPR 9 CJ 2 OPR 16 LDL 3 LDNLP 8
   LDL 1 OPR 9 CJ 2 OPR 16)

  (LDL 3 LDNL 0 LDL 3 STNL 4
   LDL 3 LDNL 1 LDL 3 STNL 5
   LDL 3 LDNL 2 LDL 3 STNL 6
   LDL 3 LDNL 3 LDL 3 STNL 7
   LDC 4 LDL 0 LDNL 4 OPR 6
   LDC 3 LDL 3 STNL 6
   LDC 3 LDL 3 STNL 7
   LDC 4 LDL 0 LDNL 5 OPR 6
   LDL 3 LDNL 4LDL 3 STNL 0
   LDL 3 LDNL 5 LDL 3 STNL 1)

  (LDL 5 LDNLP -2 STL 5 LDL 5 LDNL 1 STL 3 LDL 5 LDNL 0 OPR 6))
```

We leave out the final step generating a Lisp representation of the binary code shown above. It is easy to check that the TASM mnemonics have been correctly transformed to instruction byte sequences (cf. figure 5). We want to note, however, that in order to understand the code semantically, we additionally would have to know the semantics for instance of the Transputer operations called by OPR instructions. We *need not* know this information to perform the syntactical checking; the correctness of the code follows from compiling verification, whereas we have to check the code for compliance with the specification. We only need to



Direct Operation Codes

0x	0000	j	jump
1x	0001	ldlp	load local pointer
2x	0010	pfix	prefix
3x	0011	ldnl	load non local
4x	0100	ldc	load constant
5x	0101	ldnlp	load non local pointer
6x	0110	nfix	negative prefix
7x	0111	ldl	load local
8x	1000	adc	add constant
9x	1001	call	call
Ax	1010	cj	conditional jump
Bx	1011	ajw	adjust workspace
Cx	1100	eqc	equal to constant
Dx	1101	stl	store local
Ex	1110	stnl	store non local
Fx	1111	opr	operate

Fig. 5. Transputer Architecture and Direct Function Codes. The Transputer state consists of the registers Areg, Breg and Creg, which form a mini stack with top Areg, the operand register Oreg, the instruction pointer (program counter) Iptr, the workspace pointer Wptr, various flags like the ErrorFlag, some more registers and the memory Mem. The registers contain Word valued quantities. The memory is byte or word addressable

know 16 mnemotechnical instructions, their mapping to instruction code nibbles, and the pfix/nfix-chains necessary to load large operands.

Final Remarks

Our most recent compiler implementation consists of 237 Lisp-functions which use to be fairly small. The complete printed output for the double-checking as shown above for the largest function covers about 14 pages of program text. Only very few functions (13) need substantially more than one page.

A more sophisticated *diagonal* argument allows for saving a lot of work while checking especially low level code. We make use of the fact that the compiler is applied to itself. It runs in separate phases, and most of its code is only used in one of the phases and thus only has to be checked down to its own target code. Lower level implementations will be generated by already verified parts of the compiler. Following the strategy to do less complex tasks in lower level translation phases allows us to save much of the code inspection work. [13, 7, 19].

According to J. B. Goodenough und S. L. Gerhart [12] we can ask the question, under which as weak as possible conditions for m_C a successful strong compiler test would imply full correctness of m_C. It is interesting to note that the answer is nearly the same as in the diagonal method: Only the *diagonal* compiler phases have to be correct. [19, 18].

Our technique happens to work in our particular setting, where everything fits together quite well. Let us finally note some properties which should hold for the code generation scheme and for the choice of intermediate languages in order to keep the implementation correctness proof manageable:

- source and target code of single compilation steps should be close,
- a single compilation step should cope only with one crucial transformation,
- compilation steps should become smaller and easier towards the low level (machine code) end,
- code generation should be compositional, without optimization.

Macro-expansion like code generation allows to identify corresponding code without too much global information, and to determine the compilation rules from the context of corresponding pieces of code. Since standard coded standard hardware runs fast enough, we need no sophisticated optimization.

6 Related Work

Chirica and Martin [3] first explicitly distinguish compiling specification correctness from compiler implementation correctness. W. Polak [26] uses the Stanford-Pascal-Verifier and proves the correctness of a compiler implemented in Pascal for a Pascal like language into idealized stack code, not yet distinguishing these two tasks. J S. Moore [21] uses the Boyer/Moore prover **nqthm** to prove the correctness of a compilation function from Piton into FM8501-Code, but not the correctness of its implementation in machine code, as Moore explicitly remarks.

More recently, the VLisp project reports [24] also express the necessity of proving the compiler implementation correct. In the VLisp project, however, this task has explicitly been left out.

Our work is very closely related to Paul Curzon's work on compiler verification [4]. The crucial difference is, that he uses and trusts a theorem prover (HOL) both to carry out the proofs and to execute the compiling specification. We use PVS to mechanically support our proofs. However, no one can guarantee today, that provers are more trustworthy than compilers. The execution of the compiling specification within a prover, which of course is also a proof by itself, can be proof-checked, if necessary. In this case, however, we have either a new implementation correctness problem, or we have to check the proof protocol by hand, which is probably substantially bigger than our code checking protocol. Note that, in particular in the area of security, we are not only interested in avoiding bugs. We want to guarantee correctness of the generated code.

The idea of à posteriori double checking is related to the notion of *simple result checkers* [1, 28], which are efficient algorithms checking input/output pairs for a given program. Just inserting result checking code into the program to be checked, however, does not help, because we additionally would have to verify and also to double check this code. But it might very well divide code into critical and non-critical parts [8] and hence reduce the amount of necessary code inspection, and it may even help for verification.

7 Conclusions

We have described our approach to completely verify specification and implementation of a realistic initial fully correct compiler executable. A lot of experiments had been necessary. The subset ComLisp of Common Lisp has carefully been selected as a bootstrapping kernel, as both source and implementation language. Additionally, its applicative part (the pure functional sub-language of ComLisp) also coincides with the logic of the new Boyer/Moore prover ACL2 [17]. This links mechanical program verification to the work described here [5] and allows for proving the (partial) correctness of *the real thing*.

For the initial implementation correctness proof we use a very specialized technique in a very particular situation. This technique is not intended to be generalizable to the correct implementation of real world optimizing compilers. Once we have one initial fully correct compiler available, it can be used for correct implementation without any further explicit implementation correctness proof. The compiler guarantees sufficient (partial) correctness of executables, which is essential for the correctness of binary implementations of verified components and tools within the *Verifix* compiler verification project, in particular for generating further correct compiler or compiler-generator implementations.

Our major goal has been to provide a sound basis of a bootstrapping process for further compiler implementations, and in the *Verifix* project we will demonstrate that our techniques are repeatable. Our method allows to start fully correct compiler bootstrapping from the scratch whenever necessary. Moreover,

much of the proof work, even the code checking in the machine independent part can indeed be reused without change; we just have to check it for syntactical identity.

We have implemented the ComLisp compiler as a ComLisp program. The compiler has been bootstrapped successfully on a Transputer T400 single board computer with 1 MB of memory. The output for the proof by double checking the results has been generated. The efficiency of the generated code is acceptable: the complete bootstrap takes about one hour, mostly because all input/output-operations for every intermediate program representation have to be piped character by character through one Transputer link. We have also implemented code-generators for i386, DEC α, and MC 68000 processors, as well as for concrete C and Forth.

References

1. Manuel Blum and Sampath Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, January 1995.
2. Bettina Buth and Markus Müller-Olm. Provably Correct Compiler Implementation. In *Tutorial Material – Formal Methods Europe '93*, pages 451–465, Denmark, April 1993. IFAD Odense Teknikum.
3. L.M. Chirica and D.F. Martin. Toward Compiler Implementation Correctness Proofs. *ACM Transactions on Programming Languages and Systems*, 8(2):185–214, April 1986.
4. Paul Curzon. The Verified Compilation of Vista Programs. Internal Report, Computer Laboratory, University of Cambridge, January 1994.
5. Wolfgang Goerigk. An Exercise in Program Verification: The ACL2 Correctness Proof of a Simple Theorem Prover Executable. *Verifix*-Arbeitsbericht Verifix/CAU/2.4, CAU Kiel, 1996.
6. Wolfgang Goerigk. A Denotational Semantics for ComLisp and SIL. *Verifix*-Arbeitsbericht Verifix/CAU/2.8, CAU Kiel, December 1997.
7. Wolfgang Goerigk, Axel Dold, Thilo Gaul, Gerhard Goos, Andreas Heberle, Friedrich W. von Henke, Ulrich Hoffmann, Hans Langmaack, Holger Pfeifer, Harald Ruess, and Wolf Zimmermann. Compiler Correctness and Implementation Verification: The *Verifix* Approach. In P. Fritzson, editor, *Proceedings of the Poster Session of CC '96 – International Conference on Compiler Construction*, IDA Technical Report LiTH-IDA-R-96-12, Linkøping, Sweden, 1996.
8. Wolfgang Goerigk, Thilo Gaul, and Wolf Zimmermann. Correct Programs without Proof? On Checker-Based Program Verification. In *Proceedings ATOOLS'98 Workshop on "Tool Support for System Specification, Development, and Verification"*, Advances in Computer Science, Malente, 1998. Springer Verlag. To Appear.
9. Wolfgang Goerigk and Ulrich Hoffmann. The Compiler Implementation Language ComLisp. *Verifix*-Arbeitsbericht Verifix/CAU/1.7, CAU Kiel, June 1996.
10. Wolfgang Goerigk and Ulrich Hoffmann. The Compiling Specification from ComLisp to Executable Machine Code. Technical Report Nr. 9713, Institut für Informatik, CAU, Kiel, December 1998. In Preparation.
11. Wolfgang Goerigk and Markus Müller-Olm. Erhaltung partieller Korrektheit bei beschränkten Maschinenressourcen. – Eine Beweisskizze –. *Verifix*-Arbeitsbericht Verifix/CAU/2.5, CAU Kiel, 1996.

12. J.B. Goodenough and S.L. Gerhart. Toward a Theory of Test Data Selection. *SIGPLAN Notices*, 10(6):493–510, June 1975.
13. Ulrich Hoffmann. Über die korrekte Implementierung von Compilern. In *Workshop "Alternative Konzepte für Sprachen und Rechner"*, Bad Honnef, 1996. Auch erschienen als *Verifix*-Arbeitsbericht Verifix/CAU/3.1.
14. Ulrich Hoffmann. Korrekte Implementierung von Übersetzungsspezifikationen in hoher Programmiersprache. In Herbert Kuchen, editor, *Arbeitstagung Programmiersprachen*, volume 58 of *Arbeitsbericht des Institutes für Wirtschaftsinformatik*. Westfälische Wilhelms-Universität Münster, 1997.
15. Inmos Limited. *Transputer instruction set: A compiler writer's guide*, 1988.
16. C. B. Jones. *Systematic Software Development Using VDM, 2nd ed.* Prentice Hall, New York, London, 1990.
17. M. Kaufmann and J S. Moore. Design Goals of ACL2. Technical Report 101, Computational Logic, Inc., August 1994.
18. H. Langmaack. Ein Beitrag zu Goodenoughs und Gerharts Theorie des Softwaretestens und -verifizierens: Beziehungen zwischen starkem Übersetzertest und Übersetzerimplementierungsverifikation. *Foundations of Computer Science: Theory, Cognition, Applications, Springer Verlag.*, 1996. In Preparation.
19. Hans Langmaack. Softwareengineering zur Zertifizierung von Systemen: Spezifikations-, Implementierungs-, Übersetzerkorrektheit. *Informationstechnik und Technische Informatik it-ti*, 1997.
20. Dagobert Michelsen. Korrektheit der Daten- und Operationsverfeinerung für eine applikative Sprache mit automatischer Speicherbereinigung. Master's thesis, Institut für Informatik, CAU, Kiel, 1998.
21. J S. Moore. Piton: A verified assembly level language. Technical Report 22, Comp. Logic Inc, Austin, Texas, 1988.
22. Markus Müller-Olm. Three Views on Preservation of Partial Correctness. *Verifix*-Arbeitsbericht Verifix/CAU/5.1, CAU Kiel, October 1996.
23. Markus Müller-Olm. *Modular Compiler Verification*, volume 1283 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1997.
24. Dino P. Oliva and Mitchell Wand. A Verified Compiler for Pure PreScheme. Technical Report NU-CCS-92-5, Northeastern University College of Computer Science, Northeastern University, February 1992.
25. Holger Pfeifer, Axel Dold, F.W. von Henke, and Harald Rueß. Mechanized Semantics of Simple Imperative Programming Constructs. Ulmer Informatik-Berichte 96-11, Universität Ulm, December 1996.
26. W. Polak. Compiler specification and verification. In J. Hartmanis G. Goos, editor, *Lecture Notes in Computer Science*, number 124 in LNCS. Springer-Verlag, 1981.
27. Ken Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 27(8):761–763, 1984. Also in ACM Turing Award Lectures: The First Twenty Years 1965-1985, ACM Press, 1987, and in Computers Under Attack: Intruders, Worms, and Viruses Copyright, ACM Press 1990.
28. Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, November 1997.

Translation Validation: From DC+ to C *

A. Pnueli, O. Shtrichman, and M. Siegel

Weizmann Institute of Science, Rehovot, Israel

Abstract. *Translation validation* is an alternative to the verification of translators (compilers, code generators). Rather than proving in advance that the compiler always produces a target code which correctly implements the source code (compiler verification), each individual translation (i.e. a run of the compiler) is followed by a validation phase which verifies that the target code produced on this run correctly implements the submitted source program. In order to be a practical alternative to compiler verification, a key feature of this validation is its *full automation*.
In this paper we demonstrate the feasibility of translation validation for industrial code generators from DC+ -a widely used intermediate format for synchronous languages- to C. We explain the compilation pattern from DC+ to C and advocate new abstraction techniques for a fragment of first order logic as part of the automation of our approach.

1 Introduction

Compiler verification is an extremely complex task and every change to the compiler (even minor revisions) requires redoing the proof. Thus, compiler verification tends to "freeze" the compiler design and discourages any future improvements and revisions which is not acceptable in an industrial setting. This drawback can be avoided by a well designed translation validation approach, first introduced in [11], which compares the input and the output of the compiler for each individual run independently of *how* the output is generated from the input.

In this paper we consider translation validation for synchronous languages. Synchronous languages [8], such as Esterel [3], Argos [9], Signal [2] and Lustre [5], are mainly used in industrial applications for the development of safety critical, reactive systems. In particular, they are designed to be translatable into code which is as time/space efficient as handwritten code. This code is generated by sophisticated code generators which perform various analyses/calculations on the source code in order to derive highly efficient implementations in languages such as C and ADA. In order to share code generation tools (and silicon compilers, simulators, verification tools etc.) for synchronous languages, the DC+ format has been developed. DC+ [7] is an equational representation for both imperative and declarative synchronous languages.

* This research was done as part of the ESPRIT project SACRES and was supported in part by a grant from the Deutsche Forschungs Gemeinschaft, the Minerva Foundation and an infra-structure grant from the Israeli Ministry of Science and Art

In this paper we explain the theory underlying (fully automatic) translation validation for two industrial compilers from DC+ to C. These compilers – which apply more than 100 optimization rules during code generation [12] – are developed in the ESPRIT project SACRES by the French company TNI and by Inria (Rennes) and are used by Siemens, SNECMA and British Aerospace. Their formal verification is prohibitive due to their size (more than 20.000 lines of code each) and the fact that they are constantly improved/extended.

While developed in the context of code generators for synchronous languages, the proposed method has wider applicability. The main feature which enables us to perform the validation task algorithmically is that the source language has a restricted explicit control structure. This is also represented by the fact that the resulting C-code consists of a single main loop whose body is a loop-free program. Source languages with these features can benefit from the method proposed in this paper.

We present a common semantic model for DC+ and C, introduce the applied notion of being a correct implementation, formulate the correctness of the generated C code as proof obligations in first order logic and present efficient decision procedures to check the correctness of the generated proof obligations. All translations and constructions which are presented in the course of the paper have been implemented in a tool called TVT (Translation Validation Tool). Recently we used TVT to validated the code generated from a 6000 lines DC+ program with almost 2000 variables. This program is an turbine-control system which was developed as an industrial case study by SNECMA in the SACRES project. A report on translation validation for this case studies is in preparation.

A major advantage of a carefully designed translation validation tool is that it can replace the need for correctness proofs for *various* compilers if these compilers are based on the same definition of "correct code generation". This is the case for the TNI and the Inria compiler and, indeed, TVT is used to validate code originating from either of these two compilers.

The paper is organized as follows. In Section 2 we give a brief introduction to DC+. Section 3 presents the concepts which underly the generation of the proof obligations. In Section 4 we present the decision procedures to check the validity of these proof obligations. Section 5 contains some conclusions. An extended abstract of this paper, without the recent validation results of the turbine-control system, appeared in [10].

2 The DC+ Format

A DC+ program describes a reactive system whose behavior along time is an infinite sequence of *instants* which represent reactions, triggered by external or internal events. The main objects manipulated by a DC+ program are *flows*, which are sequences of values synchronized with a *clock*. A flow is a typed object which holds a value at each instant of its clock. The fact that a flow is currently absent is represented by the bottom symbol $\perp$ (cf. [2]). Clocks are boolean flows, assuming the values $\{\mathrm{T}, \perp\}$. A clock has the value T if and only if the flow

associated with the clock holds a value at the present instant of time. Actually, any expression *exp* in the language has its corresponding clock *clk*(*exp*) which indicates whether the value of the expression at the current instant is different from $\perp$.

Besides *external* flows (input/output flows), which determine the interface of the DC+ program with its environment, also *internal* flows are used and manipulated by the program.

2.1 DC+ and its Semantics

In order to present the formal semantics of DC+ we introduce a variant of *synchronous transition systems* (STS) [11]. STS is the computational model of our translation validation approach.

Let V be a set of typed variables. A *state* s over V is a type-consistent interpretation of the variables in V. Σ_V denotes the set of all states over V. A *synchronous transition system* $A = (V, \Theta, \rho)$ consists of a finite set V of typed variables, a satisfiable assertion Θ characterizing the initial states of system A, and a transition relation ρ. This is an assertion $\rho(V, V')$, which relates a state $s \in \Sigma_V$ to its possible successors $s' \in \Sigma_V$ by referring to both unprimed and primed versions of variables in V. Unprimed variables are interpreted according to s, primed variables according to s'. To the state space of an STS A we refer to as Σ_A. We will also use the term "system" to abbreviate "synchronous transition system". Some of the variables in V are identified as *volatile* while the others are identified as *persistent*. Volatile variables represent flows of DC+ programs, thus their domains contain the designated element $\perp$ to indicate absence of the respective flow.

A *computation* of $A = (V, \Theta, \rho)$ is an infinite sequence $\sigma = \langle s_0, s_1, s_2, \ldots \rangle$, with $s_i \in \Sigma_V$ for each $i \in \mathbf{N}$, which satisfies $s_0 \models \Theta$ and $\forall i \in \mathbf{N}.\ (s_i, s_{i+1}) \models \rho$. Denote by $||A||$ the set of computations of the STS A.

For the purpose of translation validation, DC+ programs are translated into the STS formalism. A brief introduction to DC+ and details of its translation to STS are given next.

A DC+ program consists of a set of constraints which determine the transition relation of the system. At each instant of time all constraints have to be satisfied by the values that the flows have at this instant. The constraints are expressed as *equation* and *memorization* statements. The equation $v = exp$ [1] defines the flow v to be equal to the expression *exp* at any instant, which implies that also their clocks coincide. Formally this equation contributes the following clause to the transition relation of the STS which represents the DC+ source:

$$v' = \text{if } clk(exp') \text{ then } exp' \text{ else } \perp .$$

The second kind of constraints are memorization statements $r \overset{m}{=} exp$ which defines r to hold the last (*not* including the present) non-bottom value of *exp*.

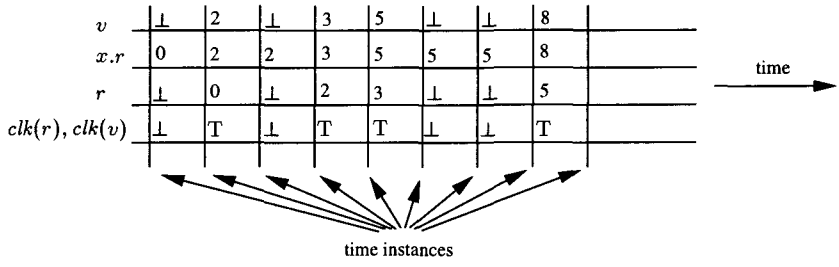
[1] we omit an optional activation condition to simplify the presentation

Also memorizations imply that the arguments have the same clocks. Whereas equations are used to specify *instantaneous reactions* of the system, memorizations are used to define the *internal state* of the system, i.e. its registers when the DC+ program is considered as an operator network [7]. The formal semantics of memorizations is

$$\begin{aligned} \wedge\ x.r' &= \text{if } clk(exp') \text{ then } exp' \text{ else } x.r \\ \wedge\ r' &= \text{if } clk(exp') \text{ then } x.r \text{ else } \bot\ . \end{aligned}$$

This definition introduces an auxiliary variable $x.r$ which stores the last (including the present) non-bottom value of *exp*. Variable $x.r$ is initialized in Θ of the STS representing the DC+ source to define the first non-bottom value of r (an init-construct in DC+ defines such initial values). From now on we refer to flows defined by memorizations as *register flows*. Variables in an STS which represent register flows will typically be denoted by r, corresponding memorization variables by $x.r$.

Example 1. The figure below shows a possible scenario for the memorization statement $r \overset{m}{=} v$.



Note, that variable $x.r$ is needed since the last non-bottom value of v is not accessible because v is volatile. Variable $x.r$ itself is persistent.

There are two kinds of functions which can be used in DC+ expressions: *monochronous* functions, such as $+, -, div, \ldots$, are standard operators on flows whose results share the same clock as their arguments while *polychronous* functions, such as $\mathsf{when}(w, cond)$ and $\mathsf{pcond}(cond, exp_1, exp_2)$, introduce and handle flows with different clocks. The latter operators can be used for under/oversampling of flows. They are translated as follows:

$$\mathsf{when}(exp, cond) \overset{\text{def}}{=} \text{if } cond = \mathrm{T} \text{ then } exp \text{ else } \bot$$

$$\mathsf{pcond}(cond, exp_1, exp_2) \overset{\text{def}}{=} \left(\begin{array}{lll} \text{if} & & cond = \mathrm{T} \wedge clk(exp_1) \text{ then } exp_1 \\ \text{else} & \text{if} & cond = \mathrm{F} \wedge clk(exp_2) \text{ then } exp_2 \\ \text{else} & \bot & \end{array} \right)$$

Based on these definitions we can define the semantics of a DC+ program D by an STS $S = (V, \Theta, \rho)$ as follows. Set V is identical to the set of flows in D plus the memorization variables $x.r$ which are introduced by the semantics above. Assertion Θ defines all variables to be initially absent [7] except memorization

variables which are initialized as stated in the DC+ source. Finally, ρ is obtained as the conjunction of the predicates which define the semantics of equation and memorization statements.

In the following sections we assume that the type definitions for variables also specify the "DC+ type" of variables, i.e. whether they are input, output, register, memorization or local variables. The respective sets of variables are denoted by I, O, R, M, L. Combinations of these letters stand for the union of the respective sets; e.g. IOR stands for the set of input/output/register variables of some system.

2.2 Compilation of Multi-clocked Synchronous Languages

The compilation scheme for multi-clocked synchronous languages (e.g. Signal, DC+) to imperative, sequential languages (e.g. C, ADA) looks as follows. The set of equation and memorization statements of a program D form a *linear equation system* LES on the flows of D and their associated clocks. Solutions of LES for a given set of input/register values determine the next state of the system. The compiler derives from D an imperative program C which consists of one main loop whose task is to repeatedly compute such solutions of the LES.

For these solutions to be efficiently computable without expensive fix-point iteration, only programs which induce an *acyclic dependency relation* on flows are compiled. This dependency relation is not static but rather depends on which flows are currently present. Thus, the compiler computes from the source program another linear equation system CC – the, so called, *clock calculus* [2] – which records the dependencies amongst clocks and a *conditional* dependency graph CDG on flows.

If CC has a unique solution for all possible inputs and CDG is acyclic, then system LES is uniquely solvable for all possible input/register values, i.e. the program D is deadlock free and determinate in its input/register variables. A formal definition of determinacy is given in Section 3.1. The program is rejected by the compiler if one of the conditions is not satisfied. Otherwise the produced code contains statements originating from the clock calculus and assignments to variables (representing the flows of D) whose order must be consistent with the dependency graph CDG. These assignments are performed if the corresponding flow is currently present in the source program, i.e. the clocks of flows determine the control structure of the generated program.

Example 2. A part of the (non-optimized) translation of a DC+ program into C code is given below:

```
.....
r_in $\stackrel{m}{=}$ in
out = when(2*r_in,in>10)
.....
```

```
WHILE true DO
{read(in);
 c_out = (in>10);
 IF c_out {out = 2*r_in;
           write(out);}
 r_in = in; }
```

For the translation validation process also the C programs are translated into the STS formalism. Since the generated C code uses in the body of the main loop

only a small fragment of ANSI C (e.g. no pointers, no loops), the translation is straightforward. Note, however that the C programs use *persistent* variables (i.e. variables which are never absent) to implement DC+ programs which use volatile variables. This has to be taken into account when defining the notion of "correct implementation" in the next section.

3 Correct Implementation: Refinement

Our approach to establish that "the C-code correctly implements the DC+ source" is based on the notion of *refinement*. The presented concepts have been approved by TNI and Inria.

3.1 Refinement and Refinement Mappings

Consider the two STSs $A = (V_A, \Theta_A, \rho_A)$ and $C = (V_C, \Theta_C, \rho_C)$, with $IO_A = IO_C$, to which we refer as the *abstract* and *concrete* system, respectively. We say that C *refines* A, denoted by C ref A, if for any $\sigma = \langle s_0, s_1, s_2, \ldots \rangle$ in $||C||$ there exists a $\tau = \langle t_0, t_1, t_2, \ldots \rangle$ in $||A||$ such that

$$\forall x \in IO_A . \forall i \in \mathbf{N}.\ s_i[x] = t_i[x] \text{ or } t_i[x] = \perp.$$

In order to establish this notion of refinement for two given systems we have to *construct* for each concrete computation $\sigma \in ||C||$ the corresponding abstract computation $\tau \in ||A||$ such that the above property is satisfied. Such constructions are usually done by means of *refinement mappings* [1]. Rather than the standard static correspondence between concrete and abstract variables, we need a more general mechanism which relates *persistent* variables of the STS-representation of the C-code (denoted C-STS from now on) to *volatile* variables of the STS-representation of the DC+ program (DC+STS).

Definition 1. *Given systems* $A = (V_A, \Theta_A, \rho_A)$ *and* $C = (V_C, \Theta_C, \rho_C)$ *with* $IO_A = IO_C$. *A mapping* $f : \Sigma_C \to \Sigma_A$ *is a* clocked refinement mapping *from* C *to* A *if it satisfies the requirements of*

- *Initiation:* $s \models \Theta_C$ *implies* $f(s) \models \Theta_A$, *for all* $s \in \Sigma_C$.
- *Propagation:* $(s, s') \models \rho_C$ *implies* $(f(s), f(s')) \models \rho_A$, *for all* $s, s' \in \Sigma_C$.
- *Preserving Observation:* $\forall x \in IO_A . \forall s \in \Sigma_C.\ f(s)[x] = s[x]$ *or* $f(s)[x] = \perp$.

The idea of this definition is, that in each time instant and for each observable variable $x \in IO_A = IO_C$ either x is present in the abstract system and $f(s)[x]$ coincides with $s[x]$ or x is absent in $f(s)$. In the following presentation we omit the qualifier "clocked" if it is clear from the context.

Theorem 1. *If there exists a clocked refinement mapping from* C *to* A *then* C ref A.

Usually, finding such a mapping f is left to the ingenuity of the verifier. In the context of translation validation it is essential that f can be *automatically constructed* from the source and target programs.

The main idea in [11] was to generate refinement mappings which reconstruct the values of *all* abstract variables. In order to do so, it was necessary to extract from the structure of the C-code the information whether an abstract variable is currently present/absent, i.e. we reconstructed the clocks of these variables. With this information about clocks we could define the correct values of abstract variables from the values of their concrete counterparts. Such a reconstruction of all abstract variables is not possible in the case of the optimizing code generators, because:

1. Internal abstract variables are possibly eliminated for space efficiency during compilation; so there are no corresponding variables in the C-code from which we could automatically reconstruct their values.
2. The reconstruction of the clocks of abstract variables was based in [11] on the assumption that an abstract variable is present iff the corresponding concrete variable has been updated in the current iteration (cf. Fig. 2). The optimizing compilers *move assignments* between if-blocks in the C-code such that neither the fact that a concrete variable is written implies that its abstract counterpart is actually present nor does the presence of an abstract variable implies that its concrete counterpart is written in the current iteration.

Since the code generators cannot eliminate IOR variables without producing incorrect code, we can exploit the property of *determinacy* – which is a central property of synchronous programs [2] – to *implicitly* reconstruct local abstract variables.

Definition 2. *An* STS $S = (V, \Theta, \rho)$ *is* determinate in $V_D \subseteq V_S$ *if:*

$$\forall s_1, s_2, s_3 \in \Sigma_S.\ ((s_1, s_2) \models \rho \wedge (s_1, s_3) \models \rho \wedge s_2[V_D] = s_3[V_D]) \Rightarrow s_2[V_S] = s_3[V_S]$$

Determinacy of S in V_D says that, after a transition, the values of variables in set $V_S \setminus V_D$ are uniquely determined once the values for the variables in V_D have been fixed. The considered compiler exclusively accept DC+ programs which are determinate in their IRM variables. Determinacy of DC+ programs is assumed from now on.

In order to determine corresponding abstract states it thus suffices to reconstruct these IRM variables by the refinement mapping. Besides this we have to reconstruct the values of abstract output variables to check whether the generated abstract and concrete outputs indeed coincide. For these IORM variables the clock generation scheme as presented in [11] can still be applied[12].

The reason is that a "re-timing" of IOR variables (i.e. moving assignments to other if-blocks as explained in point 2) tends to an incorrect implementation since either inputs/register variables would be provided at wrong time instants or/and outputs would be generated at instants where they are not supposed to occur. For memorization variables no clock has to be constructed since they always carry non-bottom values by definition.

Technically we eliminate all local variables in DC+STS $= (V, \Theta, \rho)$ by removing them from V, removing their initializations from Θ and hiding them from ρ by existential quantification. The result of applying this transformation to some STS A is denoted by $A^{\exists}$. Determinacy of A in IRM_A implies that it suffices to

construct an inductive refinement mapping from C to $A^{\exists}$ to actually prove that C correctly implements A, i.e. C ref A.

However, there is one remaining problem with the reconstruction of the values of register variables. Registers are updated during one iteration of the main loop *after* they have been used in assignments of other variables, cf. Fig. 2. Thus, at the end of an iteration, register variables are already updated for the *next* iteration. So, the values of abstract register variables have to be reconstructed from the values of the corresponding variables at the *beginning* of the iteration while input/output/memorization variables can be reconstructed from the values of corresponding variables at the *end* of the iteration. This situation is handled by automatically inserting a *history variable* $h.r$ into the C-code for each register variable r.

The constructions explained above (introduction of history variables for register variables, compression of ρ_c, hiding of local variables in ρ_a) allow us to use refinement mappings as introduced in Definition 1 to establish the correctness of the generated C-code w.r.t. the DC+ source program.

3.2 Syntactic Representation and Proof Rule for Refinement

In the quest for automating the translation validation process, we present in this section a *syntactic representation* of clocked refinement mappings and an associated proof rule. Then, we describe how the components used in the proof rule can be computed, so that the translation validation process can be carried out fully automatically.

Consider two STSs A and C with $IO_A = IO_C$. Let $\alpha : V_A \longrightarrow \mathcal{E}(V_C)$ be a *substitution* that replaces each abstract variable $v \in V_A$ by an expression $\mathcal{E}_v$ over the concrete variables V_C. Such a substitution α induces a mapping between states, denoted by f_α. For $s_C \in \Sigma_C$ the abstract state $s_A \stackrel{\text{def}}{=} f_\alpha(s_C)$ corresponding to s_C under substitution α assigns to each variable $v \in V_A$ the value of expression $\mathcal{E}_v$ evaluated in s_C. In this way, a refinement mapping can be syntactically defined by means of an appropriate substitution α. Such a substitution α is defined to be *observation preserving* if $\forall v \in IO_A.\ \models (v)[\alpha] = v \vee (v)[\alpha] = \bot$, cf. Definition 1.

Let $\alpha : V_A \longrightarrow \mathcal{E}(V_C)$ be an observation preserving substitution

R1. $\Theta_C \Rightarrow \Theta_A[\alpha]$ Initiation

R2. $\rho_C \Rightarrow \rho_A[\alpha]$ Propagation

C ref A

Rule ref: Proving Refinement

Note, that no auxiliary invariant is needed in **R2.** since code generators can not exploit reachability information for optimizations.

In order for rule ref to be useful in a fully automatic translation validation process, an appropriate substitution α has to be generated automatically.Based on the previous explanations we can define the following generic substitutions α.

Definition 3. *Given $A \in$ STS, representing the* DC+ *program where local variables have been eliminated, and $C \in$ STS, representing the C-code. We define $\alpha : V_A \longrightarrow \mathcal{E}(V_C)$ by:*

$$\begin{array}{lll} \alpha(v) = \textit{ if } \mathrm{clk}_c(v) \textit{ then } v \textit{ else } \bot & & \textit{for all } v \in IO_A (= IO_C) \\ \alpha(r) = \textit{ if } \mathrm{clk}_c(r) \textit{ then } h.r \textit{ else } \bot & & \textit{for all } r \in R_A (= R_C) \\ \alpha(x.r) = r & & \textit{for all } x.r \in M_A \end{array}$$

This specific definition of α automatically yields observation preserving substitutions.

The algorithm for computing the clock expressions above works as follows. The construction is based on viewing the body of the main loop of the C-code as a (cyclic) directed graph where every edge e is labeled by either a *guard* $\gamma(e)$ (originating from if-conditions) or an action which can be a *read* of an input variable, a *write* of an output variable, or an assignment to a variable (cf. Example 2).

The clock expression $clk_c(v)$ is computed by considering the guards along paths leading to assignments to v (resp. read/write statements in case of input/output variables). Let $path(v)$ be the conjunction of all guards along a fixed path leading to an assignment of v and let $written(v)$ be the disjunction of the predicates $path(v)$ for all possible paths leading to an assignment of v. Then $clk_c(v)$ is obtained from $written(v)$ by replacing all register variables r by $h.r$ (recall that the register variables are already updated for the next iteration). The combination of the techniques and constructions mentioned above allow us to automatically extract two first order logic formulas (corresponding to **R1.** and **R2.** in rule ref) which state the correctness of the generated code if these formulas can be shown to be valid.

The presented approach is immune against the optimizations performed by the industrial code generators that we consider. The proof technique exploits, in contrast to our previous work [11], only minimal knowledge about the code generation process. We only assume that $IORM$ variables are reconstructible which is the minimal requirement for the C-code to be a correct implementation of the DC+ source [12].

4 Checking the Proof Obligations

The generated proof obligations are infinite state assertions, thus just invoking a BDD-based decision procedure obviously does not work. Directly supplying them to a theorem prover such as PVS and starting proof strategies turned out to be far too slow.

In this section we explain the theoretical basis for an efficient BDD-based validation of the proof obligations on the basis of *uninterpreted functions*. The introduced concepts are expected to have a wider applicability as a general decision procedure for fragments of first order logic.

4.1 Preliminaries for the abstractions

All the generated proof obligations are of the form $\varphi_C \Rightarrow \exists y_1, \ldots, y_n.\ (\varphi_A \wedge \bigwedge_{i=1}^{i=n} y_i = exp_i)$ where φ_C is the left hand side of the implications in Rule ref.

The right hand side consists of the abstract local variables which are hidden by existential quantification and a conjunct φ_A which deals with the other variables. (We assume that the substitution α in Rule ref has already been performed.) In case of a determinate DC+ program, the set of equalities $y_1 = exp_1, \ldots, y_n = exp_n$ uniquely determine the values of $y_1, \ldots, y_n$ in terms of the other abstract variables. Thus we can use the following transformation in order to remove the existential quantifications from the proof obligations.

$$
\begin{array}{ll}
\varphi_C \Rightarrow \exists y.\ (y = exp \wedge \varphi_A) & \\
\varphi_C \Rightarrow \forall y.\ (y = exp \Rightarrow \varphi_A) & \text{existence and uniqueness of } y \\
\forall y.\ (\varphi_C \Rightarrow (y = exp \Rightarrow \varphi_A)) & \text{since } y \text{ does not occur free in } \varphi_C \\
\forall y.\ (\varphi_C \wedge y = exp) \Rightarrow \varphi_A & \text{propositional logic}
\end{array}
$$

This last formula is validity equivalent to the quantifier-free implication $(\varphi_C \wedge y = exp) \Rightarrow \varphi_A$. So, from now on we can concentrate on quantifier-free formulas with free variables. In order to simplify the presentation we consider formulas φ with variables of type boolean and integer and functions over these domains. Predicates are treated as boolean valued functions.

In the rest of this section we use a validity relation which is parameterized by a *declaration* D and an *interpretation* I, denoted by $\models_I^D \varphi$. Here, the declaration D determines the type of the variables in φ and I interprets a subset of the function symbols occurring in φ. We say that φ is *valid w.r.t.* (I, D), denoted by $\models_I^D \varphi$, if φ is valid in every model M where function symbols are interpreted according to I and variables according to D. Note, that M may interpret in an arbitrary way all the function symbols whose interpretation is not fixed by I.

For interpretations I_1, I_2 we define $I_1 \preceq I_2$ if I_1 and I_2 coincide on those function symbols interpreted by I_1, but I_2 possibly interprets more function symbols. Obviously, we have for $I_1 \preceq I_2$ that $\models_{I_1}^D \varphi$ implies $\models_{I_2}^D \varphi$.

The idea of the forthcoming abstractions is as follows. We have to check the validity of formula φ (the proof obligation) w.r.t. a declaration D which assigns integer/boolean types to variables and an interpretation J which gives (a standard) interpretation to all function symbols in φ.

As a sufficient condition for $\models_J^D \varphi$ we check $\models_I^D \varphi$ where $I \preceq J$ only interprets a subset of the function symbols in φ. Moving from interpretation J to I means relaxing the constraints on the interpretation of some function symbols and treating them logically as *uninterpreted.*

In a second step we apply the new technique of *function encoding*, described below, in order to substitute uninterpreted functions by fresh variables. The encoded formula, where all uninterpreted functions have been removed, is denoted by $\mathsf{F\text{-}enc}(\varphi)$ and is logically equivalent to φ under interpretation I, i.e. $\models_I^D \varphi \iff \models_I^D \mathsf{F\text{-}enc}(\varphi)$.

The encoded formulas belong to a fragment of first order logic which has a *small model* property [4]. This means that the validity of these formulas can be established by solely inspecting models up to a certain finite cardinality. In order to make these finite domains as small as possible we apply another encoding which replaces constants in $\mathsf{F\text{-}enc}(\varphi)$ by smaller constants such that the encoded formula $\mathsf{CF\text{-}enc}(\varphi)$ is logically equivalent to $\mathsf{F\text{-}enc}(\varphi)$, i.e. $\models_I^D \mathsf{F\text{-}enc}(\varphi) \iff \models_I^D \mathsf{CF\text{-}enc}(\varphi)$.

The final step of the abstraction is to determine the finite domains over which the variables of CF-enc(φ) need to be interpreted in order to faithfully check the validity of $\models_I^D$ CF-enc(φ), i.e. to determine a new declaration D^* where all variables range over finite domains such that $\models_I^D$ CF-enc(φ) $\iff \models_I^{D^*}$ CF-enc(φ) holds.

We thus get the following overall picture of our abstractions:

$$\models_J^D \varphi \quad \text{if} \quad \models_I^D \varphi \quad \text{iff} \quad \models_I^D \text{F-enc}(\varphi) \quad \text{iff} \quad \models_I^D \text{CF-enc}(\varphi) \quad \text{iff} \quad \models_I^{D^*} \text{CF-enc}(\varphi).$$

Next we explain the function encoding which is common to all our abstractions, then we illustrate the constant encoding mechanisms and the declaration changes for the individual abstractions.

4.2 The function encoding scheme

Assume we are given a formula φ, an interpretation I, and a declaration D. Furthermore, let f be a function symbol occuring in φ which is not interpreted by I. Then the function encoding scheme for f looks as follows.

- Replace each occurrence of the form $f(t_1, \ldots, t_k)$ in φ by a new variable v_f^i of a type equal to that of the value returned by f. Occurrences $f(t_1, \ldots, t_k)$ and $f(u_1, \ldots, u_k)$ are replaced by the same v_f^i iff t_j is identical to u_j for every $j = 1, \ldots, k$.
- Let $\hat{t}$ denote the result of replacing all outer-most occurrences of the form $f(t_1, \ldots, t_k)$ by the corresponding new variable v_f^i in a sub-term t of φ. For every pair of newly added variables v_f^i and v_f^j, $i \neq j$, corresponding to the non-identical occurrences $f(t_1, \ldots, t_k)$ and $f(u_1, \ldots, u_k)$, add the implication $(\hat{t_1} = \hat{u_1} \wedge \cdots \wedge \hat{t_k} = \hat{u_k}) \Rightarrow v_f^i = v_f^j$ as antecedent to the transformed formula.

Example 3. For $\varphi \stackrel{\text{def}}{=} (f(x, f(y, y)) = z \wedge x = y \wedge f(x, x) = x) \Rightarrow x = z$ the function encoding results in:

$$\left.\begin{array}{l} \wedge((x = y \wedge v_2 = y) \Rightarrow v_1 = v_2) \\ \wedge((x = x \wedge v_2 = x) \Rightarrow v_1 = v_3) \\ \wedge((y = x \wedge x = y) \Rightarrow v_2 = v_3) \end{array}\right\} \Rightarrow [(v_1 = z \wedge x = y \wedge v_3 = x) \Rightarrow x = z]$$

It is not difficult to check that both formulas are logically equivalent, indeed both are valid formulas under any declaration D and any interpretation I which does not interpret function symbol f.

Let f-enc(φ) denote the result of applying the function encoding to φ.

Theorem 2. *Given a formula φ, an interpretation I, and a declaration D. Let f be a function symbol which is not interpreted by I. Then φ is valid w.r.t. (I, D) iff* f-enc(φ) *is valid w.r.t. (I, D).*

4.3 Level-zero abstraction

In Level-zero abstraction we consider the validity of the generated proof obligation w.r.t. an interpretation I which only gives interpretations to (polymorphic)

equality, boolean functions (i.e. functions with boolean domain and range) and if-then-else. All remaining function symbols are left uninterpreted and are successively removed by the above scheme. Let F-enc(φ) denote the resulting formula after elimination.

F-enc(φ) belongs to a fragment of first order logic formulas which have a small model property [4] which we exploit to check the validity of $\models_I^D$ F-enc(φ). In order to limit the domains over which we have to interpret the integer variables in F-enc(φ) we apply a constant encoding scheme where the integer constants appearing in F-enc(φ) replaced by smaller integer constants. Such an encoding is possible since no ordering information can be expressed in the considered fragment of first order logic (note that we treat at Level-zero also comparison functions as being uninterpreted).

Let $\mathcal{C}$ denote the set of integer constants appearing in F-enc(φ), and let $|\mathcal{C}|$ denote the size of $\mathcal{C}$. Let π be any bijection from $\mathcal{C}$ to $\{0, \ldots, |\mathcal{C}|-1\}$.

The constant encoding consists of replacing each constant $c \in \mathcal{C}$ by its encoding $\pi(c)$. Let CF-enc(φ) denote the result of applying the constant encoding transformation to F-enc(φ). The following claim, where interpretation I, declaration D, F-enc(φ) and CF-enc(φ) are defined as above, justifies this encoding.

Theorem 3. *Formula* F-enc(φ) *is valid w.r.t.* (I, D) *iff* CF-enc(φ) *is valid w.r.t.* (I, D).

Finally, in order to check the validity of CF-enc(φ) w.r.t. (I, D) we alter the standard declaration D to a declaration D^* which associates *finite* types with all variables previously declared to be integers.

Let N denote the number of distinct variables appearing in CF-enc(φ) and, as before, let $|\mathcal{C}|$ denote the number of distinct integer constants appearing in CF-enc(φ). Since CF-enc(φ) has been obtained by applying the constant-encoding transformation, we know that all of these constants lie in the range $\{0, \ldots, |\mathcal{C}| - 1\}$. Let D^* denote the modified declaration in which all integer variables are redeclared to belong to the integer sub-type $\{0, \ldots, |\mathcal{C}|+N-1\}$.

In the following claim let I, D, D^* and CF-enc(φ) be as defined as above.

Theorem 4. CF-enc(φ) *is valid w.r.t.* (I, D) *iff it is valid w.r.t.* (I, D^*).

Now, validity of CF-enc(φ) w.r.t. (I, D^*) is a sufficient condition for the validity of φ w.r.t. the original interpretation J and the original declaration D. However, as in all abstraction approaches, if $\models_I^{D^*}$ CF-enc(φ) does not hold we can not conclude anything for the validity of the original formula. Thus we suggest to use a more refined abstraction if the Level-zero abstraction failed.

4.4 Level-one abstraction

In Level-one we keep the interpretation of equality, boolean functions, if-then-else and additionally of comparison operators.

The formulas F-enc(φ) – resulting from function encoding of all uninterpreted function symbols – again possess the small model property. However, a different constant encoding scheme than in the Level-zero has to be used since now ordering information amongst variables and constants can be expressed and thus has to be preserved.

Let $\mathcal{C} = \{c_1, \ldots, c_m\}$ be the set of constants appearing in F-enc(φ) where $c_1 < \cdots < c_m$. We introduce new variables $v_{c_1}, \ldots, v_{c_m}$ and transform F-enc(φ) by replacing all constant symbols by their respective new variables. Then we define for a pair (c_1, c_2), $c_1 < c_2$, of constants the following clause (N denotes again the number of distinct variables appearing in F-enc(φ)):

$$\mathsf{const}(c_1, c_2) = \begin{cases} v_{c_1} < v_{c_2} \wedge v_{c_2} - v_{c_1} \leq (c_2 - c_1) & \text{if } c_2 - c_1 \leq N \\ v_{c_2} > v_{c_1} & \text{if } c_2 - c_1 > N \end{cases}$$

Then, we add the predicate $\mathsf{const}(c_1, c_2) \wedge \ldots \wedge \mathsf{const}(c_{m-1}, c_m)$ as antecedent to the transformed formula. The result is again denoted by CF-enc(φ). Let interpretation I, declaration D, F-enc(φ) and CF-enc(φ) be as defined above.

Theorem 5. F-enc(φ) *is valid w.r.t.* (I, D) *iff* CF-enc(φ) *is valid w.r.t.* (I, D).

Finally, the standard declaration D is altered. Let N denote the number of distinct variables appearing in CF-enc(φ). Let D^* denote the modified declaration in which all integer variables are redeclared to belong to the integer sub-type $\{0, \ldots, N-1\}$. The following theorem justifies this transformation where I, D, D^*, and CF-enc(φ) are defined as above.

Theorem 6. CF-enc(φ) *is valid w.r.t.* (I, D) *iff it is valid w.r.t.* (I, D^*).

Obviously, Level-one yields more faithful abstractions than Level-zero. However, what do we do in case that also Level-one fails? Currently we are elaborating a hierarchy of abstractions by removing less interpretations of function symbols from the original formula. However, for the purpose of translation validation our experience suggests that Level-zero and Level-one will be sufficient to establish validity of the proof obligations if the generated code is indeed a correct implementation of its DC+ source.

5 Conclusion

We have presented the theory which underlies our fully automatic translation validation approach for optimizing industrial compilers from DC+ to C. The insertion of history variables into the C code, the translation of DC+ and C programs to STS, the compression of the concrete transition relation, the generation of the substitution α and the final assembling of the proof obligations according to Rule ref have all been implemented in TVT (Translation Validation Tool). TVT uses the decision procedures explained in Section 4 in order to check the validity of the generated proof obligations. A report on translation validation with TVT of the turbine-control system by SNECMA is in preparation.

Related work: In [11] we addressed translation validation for a compiler from Signal to C which did not perform optimizations while generating the C code. The revision of this work to deal with the optimizing compilers from TNI and Inria is the topic of this paper.

The work in [6] performs translation validation on a purely syntactic level. Their method is based on finding a bijection between abstract and concrete instruction sets (resp. variables) because they are considering a *structural* translation of one sequential program into another sequential program. Since we are

dealing with optimizing compilers we have to employ a purely *semantic* approach which is far more involved than finding a syntactic correspondence between programs. We are not aware of any other translation validation approaches for industrial compilers.

Future work: Currently the generated C implementations of DC+ programs are centralized. Nevertheless, the code generators that we consider are currently extended to also generate *distributed* implementations. We started to investigate the adaptation of the presented framework to also cope with distributed implementations. The main idea here is to reconstruct a centralized implementation for the distributed implementation and then to apply the presented techniques.

Furthermore we currently look for more faithful abstractions, corresponding new constant encoding schemes and better lower bounds for the finite domains over which the transformed proof obligations are interpreted.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2), 1991.

[2] A. Benviniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language. *Science of Computer Programming*, 16, 1991.

[3] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2), 1992.

[4] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer, 1996.

[5] P. Caspi, N. Halbwachs, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9), 1991.

[6] A. Cimatti, F. Giunchiglia, and P. Pecchiari et al. A provably correct embedded verifier for the certification of safety critical software. In *CAV*, number 1254 in LNCS. Springer, 1997.

[7] The declarative code DC+. ESPRIT Project: SACRES, Project Report, 1997. Version 1.3.

[8] *Another look at real-time programming*, volume 9 of *Proc. of the IEEE*, September 1991.

[9] F. Maraninchi. Operational and compositional semantics of synchronous automata compositions. In *Proceedings CONCUR*, volume 630 of *LNCS*. Springer, 1992.

[10] A. Pnueli, M. Siegel, and O. Shtrichman. Translation validation for synchronous languages. In *ICALP '98*, LNCS. Springer-Verlag, 1998.

[11] A. Pnueli, M. Siegel, and E. Singermann. Translation validation. In *TACAS 98: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS. Springer-Verlag, 1998.

[12] private communications with TNI (BREST), Siemens (Munich) and Inria (Rennes).

A Practical Hierarchical Design by Timed Simulation Relations for Real-Time Systems

Satoshi Yamane

Department of Computer Science, Faculty of Science, The University of Shimane
1060 Nishikawatu, Matsue City, Japan, 690
E-mail: yamane@cis.shimane-u.ac.jp

Abstract. As many processes concurrently behave and timing constraints are strict in real-time systems, it is difficult to design real-time systems. For this reason, a hierarchical design method is useful. In the hierarchical design method, it is important to verify whether the low level specification satisfies the high level specification or not. In general, the language inclusion verification method is useful for verifying it. But, as nondeterministic timed automata are not closed under complementation, it is impossible to use the language inclusion verification method.
In this paper, we propose the hierarchical design method based on timed simulation method. Especially, we generalize existing timed simulation methods and propose a *safety* timed simulation relation and a $\exists$-liveness timed simulation relation, a $\forall$-liveness timed simulation relation. Finally, we show our proposed method effective by some example.

1 Introduction

As many processes concurrently behave and timing conditions are strict in real-time systems, it is important to formally specify and verify real-time systems. Moreover,it is important to hierarchically specify and verify real-time systems. In this paper, we propose the following method : We specify all the hierarchies using the same specification language, and verify the consistencies between them. Especially we focus on formal specification and verification based on the notions of both time and fairness. Fairness is a mathematical abstraction by the assumption that each processor is always running at some positive [Francez 86]. Namely, fairness abstracts the details of fair schedulers and the speeds of independent processors. Especially, we can not specify communication protocols with unreliable buffers unless we assume fairness.

Formal verification methods of timed automata are almost classified into language inclusion [Alur 92] and model-checking [Alur 90], bisimulation [Milner 89]. Language inclusion and simulation relation, bisimulation relation are useful for verifying the consistencies between hierarchies.

1. We can easily and naturally verify fairness and regularity as acceptance conditions by language inclusion method. But if we specify verification properties using nondeterministic timed automata, language inclusion problems

are undecidable. Timed automata are not closed under complementation [Alur 92].

2. Bisimulation relation is useful for verifying a kind of invariant holding between hierarchies. But when we hierarchically and incrementally develop specifications, we may add exception processings to the low level specification, which are not contained in the high level specification. Bisimulation relation is not adequate for this reason.

From the above discussion, simulation relation is useful for our purpose.

Next, we survey existing studies about verifications of simulation relations as follows.

1. Lynch(MIT) has developed axioms of simulation relations for verifying distributed algorithms [Lynch 87]. Moreover, she has developed axioms of timed simulation relations for verifying real-time systems, too [Lynch 92].
2. Dill(Stanford) has extended simulation relations with fairness constraints, and has proposed the automatic verification methods using BDDs [Dill 91]. But he has not extended his simulation relation with timing constraints.
3. Tasiran(California Berkley) has proposed timed simulation relations and the hierarchical design method [Tasiran 96]. But he has not considered fairness constraints.

In this paper, we propose simulation relations with fairness and time. Moreover, we propose the following hierarchical design method : We describe all the specifications from requirement stages to design stages using nondeterministic timed automata, and verify the consistencies between all the specifications.

The remainder of the paper is organized as follows. In the next section, we introduce the specification method for real-time systems. In Section 3, we propose timed simulation relations with fairness and time, and formal verification methods based on it. In Section 4, we give some experimental results to show the effectiveness of this method. Section 5 concludes the paper.

2 Specification method of real-time systems

We specify requirements and designs using nondeterministic timed automata [Alur 92].

Definition 1 (Definition of nondeterministic timed automata) *A nondeterministic timed automaton A is a tuple $(\Sigma, S, S_0, C, E, F)$, where,*

Σ *: a finite set of events*
S *: a finite set of states*
$S_0 \subseteq S$ *: a finite set of initial states*
C *: a finite set of clocks*
$E \subseteq S \times S \times \Sigma \times 2^C \times \Phi(C)$ *: transition relations*
$F \subseteq S$ *is a finite set of Buchi accepting states*

For a set C of clock variables($x \in C$), the set $\Phi(C)$ of clock constraints δ is defined inductively :

$$\delta ::= x \leq d \mid x \geq d \mid \neg\delta \mid \delta_1 \wedge \delta_2$$

where d is a natural number.

An edge $(s, s\prime, a, \lambda, \delta)$ represents a transition from s to s$\prime$ on input symbol a. The set $\lambda \subseteq C$ gives the clocks to be reset with this transition, and δ is a clock constraint over C.

A run r of a nondeterministic timed automaton is called an accepting run iff $inf(r) \cap F \neq \phi$. □

This nondeterministic timed automaton is a powerful specification language as follows.

1. every timed ω-regular language is expressible as a nondeterministic timed automaton. But deterministic timed automata are less expressive than nondeterministic timed automata.
2. A nondeterministic timed automaton is the mathematical abstraction of a deterministic timed automaton. If we specify systems using a nondeterministic timed automaton, the number of states is less than when we specify systems using a deterministic timed automaton.

From the above, it is important to specify all the hierarchies using nondeterministic timed automata.

We can define the semantics of nondeterministic timed automata as timed ω-regular language as shown in [Alur 92].

Next, we show some example of a nondeterministic timed automaton.

Example 1 (Example of a nondeterministic timed automaton) *We show some example of a nondeterministic timed automaton in figure 1.*

When $(s, s\prime, a, \lambda, \delta) \in E$, we denote $s \xrightarrow{a,\lambda,\delta} s\prime$. An event a and a reset formula λ, a timing constraint formula δ exists on the edge.

The automaton starts in state s_0, and moves to state s_1 reading the input symbol a. The clock y gets set to 0 along with this transition. While in state s_1, the value of the clock y shows the time elapsed since the occurrence of the last a symbol. The transition from state s_1 to s_2 is enabled reading the input symbol b only if this value is equal to 1. Moreover, the transition from state s_1 to s_3 is enabled reading the input symbol b only if this value is less than 2. From these transitions, if the clock y is equal to 1 reading the input symbol b in state s_1, the transition from state s_1 to s_2 or s_3 is enabled. Namely, this timed automaton is nondeterministic. □

Real-time systems consist of a number of interacting processes. For a clear and compact description, each component can be represented with a separate nondeterministic timed automaton, and their parallel execution modeled by their automaton composition. As this composition has been reported in the paper [Alur 92], we omit it.

3 Hierarchical design method based on timed simulation relation

In this paper, we propose the hierarchical design method based on both timed simulation relations and nondeterministic timed automata. We can realize the

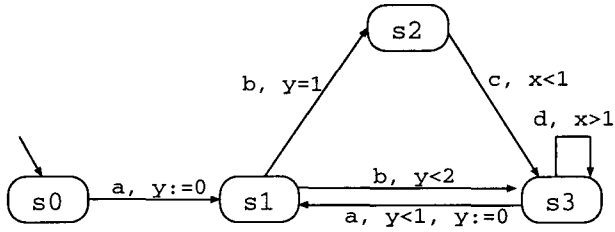


Fig. 1. Example of nondeterministic timed automaton

hierarchical design method without complementing nondeterministic timed automata.

3.1 Hierarchical design method

We specify both the high level and the low level by nondeterministic timed automata, and verify the consistencies between them by timed simulation relations.

Definition 2 (Hierarchical design method) *The hierarchical design method for real-time systems consists of the following procedures. We illustrate it in Figure 2.*

1. *First, we specify the high level by nondeterministic timed automata.*
2. *Secondly, we specify the low level based on the high level by nondeterministic timed automata.*
3. *Finally, we refine the low level until the timed simulation relations from the low level to the high level are satisfied.*
4. *We repeat the above steps, and specify the final level.*

3.2 Verification the method by timed simulation relations

First we construct region graphs, and secondly we verify timed simulation relations.

Construction of region graphs: Since the number of timed states is infinite, we cannot possibly build an automaton. But if two timed states with the same state agree on the integral parts of all clock values, and also the ordering of the fractional parts of all clock values, then the runs starting from the two timed states are very similar. From the above facts, we can construct region graphs, which are finite quotient structures by equivalence relations [Alur 92].

First, we define equivalence relations.

Definition 3 (Equivalence relations) *Let A be a nondeterministic timed automaton. For each $x \in C$,let c_x be the largest integer.*

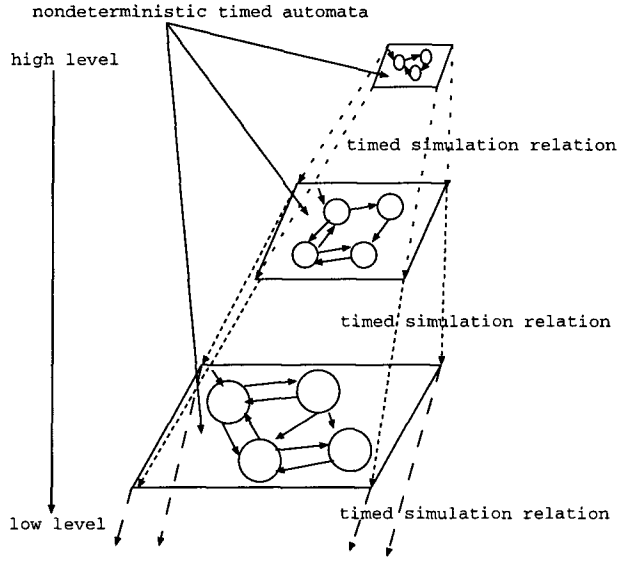


Fig. 2. Hierarchical design method

For any $t \in R$, fract(t) denotes the fractional part of t, and [t] denotes the integral part of t;i.e.,t=[t]+fract(t).

Let $\Gamma(A)$ denote $[C \to R]$, the set of time assignments for the clocks of A. Let $\nu, \nu\prime \in [C \to R]$ and $t \in R$.

The equivalence relation $\equiv$ is defined over the set of all clock interpretations for C;$\nu \equiv \nu\prime$ iff all the following conditions hold:

1. *For all $x \in C$, either $[\nu(x)]$ and $[\nu\prime(x)]$ are the same, or both $[\nu(x)]$ and $[\nu\prime(x)]$ are greater than c_x.*
2. *For all $x, y \in C$ with $\nu(x) \leq c_x$ and $\nu(y) \leq c_y$, $fract(\nu(x)) \leq fract(\nu(y))$ iff $fract(\nu\prime(x)) \leq fract(\nu\prime(y))$.*
3. *For all $x \in C$ with $\nu(x) \leq c_x$, $fract(\nu(x)) = 0$ iff $fract(\nu\prime(x)) = 0$.* □

Next, we define region graphs of nondeterministic timed automata based on the equivalence relation.

Definition 4 (Region graphs) *We define a region graph $G = (Q, Q_0, \Sigma, E\prime, F\prime)$ of a nondeterministic timed automaton A as follows.*

1. *$Q : < s, [\nu] >$ is a finite set of states*
   (a) *$s \in S$*
   (b) *$[\nu] = \{\nu\prime | \nu \equiv \nu\prime\}$*
      i. *$\nu \in \Gamma(A)$*
      ii. *$\Gamma(A) : C \to R$*
2. *$Q_0 :< s_0, [\nu_0] >$ is a finite set of initial states*
   (a) *$s_0 \in S_0$*

(b) $[\nu_0]$ *means assigning 0 to every clock*

3. Σ *: a finite set of events*
4. $E' \subseteq Q \times \Sigma \times Q$ *: state transition relations*
5. F' *: a finite set of* $< s, [\nu] >$ *for* $\forall s \in F$ □

Example 2 (Example of constructing region graphs)
We construct (2) clock regions from (1) nondeterministic timed automata, and construct (3) region graphs. We show this region graph in Figure 3. In this example, as clock constants are 0 *and* 1*, we construct the clock region in (2). From it, we construct region graphs in (3). When* $0 < z \leq 1$*, two transitions are enabled. This fact means a nondeterministic timed automaton.* □

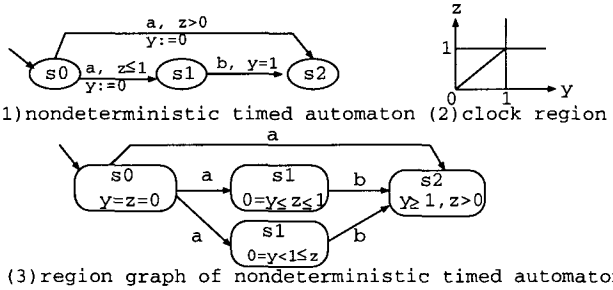


Fig. 3. Example of constructing region graphs

Timed simulation relations: First, we define the following three timed simulation relations over region graphs of nondeterministic timed automata.

1. *safety timed simulation relation* without fairness
2. $\exists$*-liveness timed simulation relation* satisfying fairness in some state sequence
3. $\forall$*-liveness timed simulation relation* satisfying fairness in all state sequences

Definition 5 (Definition of safety timed simulation relation)
$G = (Q, Q_0, \Sigma, E, F)$ *and* $G' = (Q', Q_0', \Sigma, E', F')$ *are region graphs of nondeterministic timed automata. A binary relation* $R \subseteq Q \times Q'$ *is a safety timed simulation relation from* G *to* G' *denoted by* $G \preceq G'$ *if the following conditions hold:*

1. *(safety timed simulation relation)*
   If $(q_i, q_i') \in R$ *and there exists* q_{i+1} *such that for* $\forall \sigma \in \Sigma$, $q_i \xrightarrow{\sigma} q_{i+1}$*, there exists* q_{i+1}' *such that* $q_i' \xrightarrow{\sigma} q_{i+1}'$ *and* $(q_{i+1}, q_{i+1}') \in R$*, where* $q_i, q_{i+1} \in Q$ *and* $q_i', q_{i+1}' \in Q'$ *such that* $q_i \equiv q_i'$ *and* $q_{i+1} \equiv q_{i+1}'$*. Here we denote* $< q_i, \nu_1 > \equiv < q_i', \nu_2 >$ *by extending* $\equiv$ *when* $\nu_1 \equiv \nu_2$ *holds.*

2. *(an initial condition)*
   There exists $q_0{}' \in Q_0{}'$ *satisfying* $(q_0, q_0{}') \in R$ *if* $\forall q_0 \in Q_0$. □

Definition 6 (Definition of ∃-liveness timed simulation relation) *$G = (Q, Q_0, \Sigma, E, F)$ and $G' = (Q', Q_0', \Sigma, E', F')$ are region graphs of nondeterministic timed automata. A binary relation $R \subseteq Q \times Q'$ is a ∃-liveness timed simulation relation from G to G' denoted by G $\exists$-$\preceq$ G' if the following conditions hold:*

1. *(∃-liveness timed simulation relation)*
   For every fair state sequence $q_0, q_1, \ldots$, there exists a fair state sequence $q_0', q_1', \ldots$ satisfying the following conditions:
   If $(q_i, q_i') \in R$ and there exists q_{i+1} such that for $\forall \sigma \in \Sigma$, $q_i \xrightarrow{\sigma} q_{i+1}$, then there exists q_{i+1}' such that $q_i' \xrightarrow{\sigma} q_{i+1}'$ and $(q_{i+1}, q_{i+1}') \in R$, where $q_i, q_{i+1} \in Q$ and $q_i', q_{i+1}' \in Q'$ such that $q_i \equiv q_i'$ and $q_{i+1} \equiv q_{i+1}'$. Here we denote $< q_i, \nu_1 > \equiv < q_i', \nu_2 >$ by extending $\equiv$ when $\nu_1 \equiv \nu_2$ holds.
2. *(an initial condition)*
   There exists $q_0' \in Q_0'$ satisfying $(q_0, q_0') \in R$ if $\forall q_0 \in Q_0$. □

Definition 7 (Definition of ∀-liveness timed simulation relation) *$G = (Q, Q_0, \Sigma, E, F)$ and $G' = (Q', Q_0', \Sigma, E', F')$ are region graphs of nondeterministic timed automata. A binary relation $R \subseteq Q \times Q'$ is a ∀-liveness timed simulation relation from G to G' denoted by G $\forall$-$\preceq$ G' if the following conditions hold:*

1. *(∀-liveness timed simulation relation)*
   For every fair state sequence $q_0, q_1, \ldots$, there exists every fair state sequence $q_0', q_1', \ldots$ satisfying the following conditions:
   If $(q_i, q_i') \in R$ and there exists q_{i+1} such that for $\forall \sigma \in \Sigma$, $q_i \xrightarrow{\sigma} q_{i+1}$, then there exists q_{i+1}' such that $q_i' \xrightarrow{\sigma} q_{i+1}'$ and $(q_{i+1}, q_{i+1}') \in R$, where $q_i, q_{i+1} \in Q$ and $q_i', q_{i+1}' \in Q'$ such that $q_i \equiv q_i'$ and $q_{i+1} \equiv q_{i+1}'$. Here we denote $< q_i, \nu_1 > \equiv < q_i', \nu_2 >$ by extending $\equiv$ when $\nu_1 \equiv \nu_2$ holds.
2. *(an initial condition)*
   If $\forall q_0 \in Q_0$, there exists $q_0' \in Q_0'$ satisfying $(q_0, q_0') \in R$. □

In this paper, if there exists a timed simulation relation from the low level G to the high level G', we say G satisfies G'.

Theorem 31 (Decidability of timed simulation relations) *The timed simulation relations between nondeterministic timed automata are decidable.*
(Outline of proof)
We can construct region graphs from nondeterministic timed automata [Alur 92]. On the other hand, computing simulation relations between finite state transition systems is decidable. For example, computing safety simulation relation has been reported in the paper [Cleaveland 89]. Region graphs are finite state transition systems. From these results, we can conclude the timed simulation relations between nondeterministic timed automata are decidable. □

Theorem 32 (Relations between timed simulation and language inclusion) *Timed simulation relations are stronger relations than timed language inclusion relations.*

(Proof)

Timed simulation relations can be viewed as simulation relations on region graphs. If simulation relations hold true, language inclusion relations clearly hold true, too. But even if language inclusion relations hold true, simulation relations do not hold true.

In Figure 4, we show examples of safety timed simulation relation. As Figure 4 (1) and (2) accept

$$< q_0, x = 0 > \xrightarrow{a} < q_1, 0 < x \leq 1 > \xrightarrow{b}$$

$$< q_2, x > 1 > \xrightarrow{d} < q_0, x = 0 >$$

or

$$< q_0, x = 0 > \xrightarrow{a} < q_1, 0 < x \leq 1 > \xrightarrow{c}$$

$$< q_3, x > 1 > \xrightarrow{d} < q_0, x = 0 >$$

, language inclusion relations hold true. Here $[\nu]$ means clock regions and represents inequalities. But if (1) accepts a, it accepts both b and c. On the other hand, if (2) accepts a, it accepts b or c. From these facts, language inclusion relations hold true, but simulation relations do not hold true.

We can conclude timed simulation relations are stronger relations than timed language inclusion relations. □

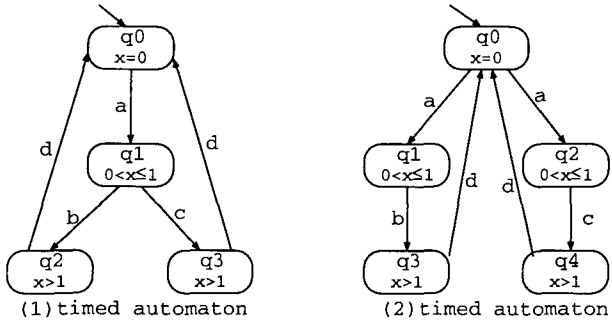


Fig. 4. Relations between safety simulation relations and language inclusio

Theorem 33 (Relations between ∃-liveness and ∀–liveness timed simulation) *∀-liveness timed simulation relations are stronger relations than ∃-liveness timed language inclusion relations.*

(Proof)
From these definitions, it is clearly true.

Next, we show some example in Figure 5, which satisfies ∃-liveness timed simulation relations but does not satisfy ∀-liveness timed simulation relations.

There exists every fair state sequence of the low level as follows :

$$< s_0, x = 0 > \xrightarrow{a}$$
$$(< s_1, 0 < x < 1 > \xrightarrow{a} < s_1, 0 < x < 1 >)^{\omega}$$
$$\xrightarrow{a} (< s_2, x \geq 1 > \xrightarrow{b} < s_2, x \geq 1 >)^{*}$$

and

$$< s_0, x = 0 > \xrightarrow{b}$$
$$(< s_1, 0 < x < 1 > \xrightarrow{a} < s_1, 0 < x < 1 >)^{\omega}$$
$$\xrightarrow{b} (< s_2, x \geq 1 > \xrightarrow{b} < s_2, x \geq 1 >)^{*}$$

and

$$< s_0, x = 0 > \xrightarrow{c}$$
$$(< s_1, 0 < x < 1 > \xrightarrow{a} < s_1, 0 < x < 1 >)^{\omega}$$
$$\xrightarrow{c} (< s_2, x \geq 1 > \xrightarrow{b} < s_2, x \geq 1 >)^{*}$$

and

$$< s_0, x = 0 > \xrightarrow{d}$$
$$(< s_1, 0 < x < 1 > \xrightarrow{a} < s_1, 0 < x < 1 >)^{\omega}$$
$$\xrightarrow{d} (< s_2, x \geq 1 > \xrightarrow{b} < s_2, x \geq 1 >)^{*}.$$

On the other hand, there exists every fair state sequence of the high level as follows :

$$< s_0, x = 0 > \xrightarrow{(a+b+c+d)}$$
$$(< s_1, 0 < x < 1 > \xrightarrow{a} < s_1, 0 < x < 1 >)^{\omega}$$
$$\xrightarrow{(a+b+c+d)} (< s_2, x \geq 1 > \xrightarrow{b} < s_2, x \geq 1 >)^{*}.$$

From these facts, there exists a ∃-liveness timed simulation relation from the low level to the high level.

But for the following state sequence :

$$< s_0, x = 0 > \xrightarrow{(a+b+c+d)}$$
$$(< s_1, 0 < x < 1 > \xrightarrow{a} < s_1, 0 < x < 1 >)^{*}$$
$$\xrightarrow{(a+b+c+d)} (< s_2, x \geq 1 > \xrightarrow{b} < s_2, x \geq 1 >)^{\omega},$$

there does not exist a ∃-liveness timed simulation relation from the low level to the high level.

From these facts, there does not exist a ∀-liveness timed simulation relation from the low level to the high level. □

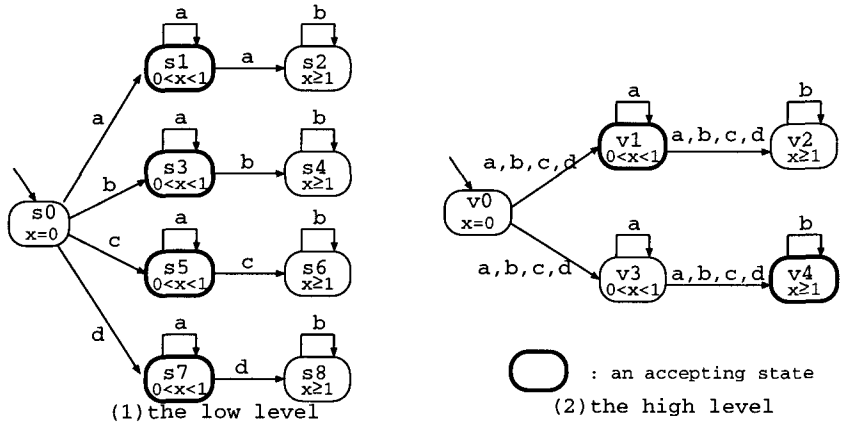


Fig. 5. Relations between ∃-liveness and ∀-liveness timed simulations

Verification algorithm: Verification algorithms based on timed simulation relations between nondeterministic timed automata consist of the following procedures. First, we construct region graphs of both the high level and the low level of nondeterministic timed automata. Next, we verify timed simulation relations between the high level and the low level of region graphs.

Definition 8 (Algorithm of a safety timed simulation relations) *The algorithm of safety timed simulation relations between nondeterministic timed automata consists of the following procedures.*

1. *We construct region graphs of both the high level and the low level of nondeterministic timed automata.*
   (a) We construct clock regions from timing constants.
   (b) We construct region graphs based on clock regions.
2. *We verify the safety timed simulation relations between the high level and the low level by the following procedure. Basically, we inductively calculate $R(k)$ such as $R(0), \ldots, R(k), R(k+1)$. Finally, when $R(k) = R(k+1)$ holds true, we decide $R = R(k)$.*
   where,
   $G = (Q, Q_0, \Sigma, E, F)$ *: the low level specification*
   $G\prime = (Q\prime, Q_0\prime, \Sigma, E\prime, F\prime)$ *: the high level specification.*

   (a) First, we define $R(0) = Q \times Q\prime$.
   (b) Next, we inductively calculate $R(k+1)(k \geq 0)$ from $R(k)$ by the following procedure.
      i. $R(k+1) = \phi$.
      ii. For every $(q_i, q_i\prime) \in R(k)$ and $\forall\sigma \in \Sigma$,
         A. If there exists q_{i+1} such that $q_i \xrightarrow{\sigma} q_{i+1}$, then there exists $q_{i+1}\prime$ such that $q_i\prime \xrightarrow{\sigma} q_{i+1}\prime$, and $(q_{i+1}, q_{i+1}\prime) \in R(k)$ holds true.

B. Both q_i and $q_i{}'$ are contained in the equivalence class of clock regions, and both q_{i+1} and $q_{i+1}{}'$ are contained in the equivalence class of clock regions.

When the previous two conditions A. and B. hold true, we calculate $R(k+1) = R(k) \cup \{(q_i, q_i{}')\}$.

iii. If $R(k+1) = R(k)$, then go to (c). If $R(k+1) \neq R(k)$, then return to (b)i. .

(c) There exists a safety timed simulation relation from G to G' iff , for $\forall q_0 \in Q_0$, there exists $q_0{}' \in Q_0{}'$ satisfying $(q_0, q_0{}') \in R(k)$.

As the previous (a) and (b), (c) are formalized as a greatest fixpoint, the algorithm stops. If there exists a safety timed simulation relation from G to G' by (a) and (b), (c), we say the low level G satisfies the high level G'. □

In a ∃-liveness timed simulation relation and a ∀-liveness timed simulation relation, we calculate the following check after we calculate a safety timed simulation relation $R(k)$.

Definition 9 (Algorithm of a ∃-liveness timed simulation relation) *The algorithm of verifying a ∃-liveness timed simulation relation consists of the following procedure.*

1. *We calculate $R(k)$ satisfying a safety timed simulation relation. Next, we calculate the product automaton $P = (W, W_0, \Sigma, N, Acc)$ of $G = (Q, Q_0, \Sigma, E, F)$ and $G' = (Q', Q_0', \Sigma, E', F')$ according to $R(k)$.*
   (a) $W = Q \times Q'$
   (b) $W_0 = Q_0 \times Q_0'$
   (c) Σ : a finite set of events
   (d) $N \subseteq W \times \Sigma \times W$: state transition relations
   (e) $Acc = F \times F'$: a finite set of accepting states
2. *We calculate the following procedure about the product automaton P.*
   (a) We calculate strong connected components both including accepting states of G and reachable from initial states.
   (b) If the strong connected components include one of accepting states of G', we say $R(k)$ is a ∃-liveness timed simulation relation. If not so, we say $R(k)$ is not a ∃-liveness timed simulation relation. □

Definition 10 (Algorithm of a ∀-liveness timed simulation relation) *The algorithm of verifying a ∀-liveness timed simulation relation consists of the following procedure.*

1. *We calculate $R(k)$ satisfying a safety timed simulation relation. We calculate the product automaton $P = (W, W_0, \Sigma, N, Acc)$ of $G = (Q, Q_0, \Sigma, E, F)$ and $G' = (Q', Q_0', \Sigma, E', F')$ according to $R(k)$.*
   (a) $W = Q \times Q'$
   (b) $W_0 = Q_0 \times Q_0'$
   (c) Σ : a finite set of events
   (d) $N \subseteq W \times \Sigma \times W$: state transition relations

(e) $Acc = F \times F'$: *a finite set of accepting states*

2. *We calculate the following procedure about the product automaton* P.
   (a) *We calculate strong connected components both including accepting states of* G *and reachable from initial states.*
   (b) *If the strong connected components include every accepting state of* G', *we say* $R(k)$ *is a* $\forall$*-liveness timed simulation relation. If not so, we say* $R(k)$ *is not a* $\forall$*-liveness timed simulation relation.* □

Example 3 (Example of verifying timed simulation relations) *(1)region graphs of the high level and (2)region graphs of the low level are given. We inductively calculate a safety timed simulation relation as follows.*

1. $R(0)$ = *all the relations*
2. $R(1) = \{(v_0, s_0), (v_1, s_1), (v_2, s_2), (v_3, s_2), (v_4, s_5)\}$

Here, as $R(1) = R(2)$ *and* $(v_0, s_0) \in R(1)$ *hold true, there exists a safety timed simulation relation from the low level to the high level. Namely, the low level satisfies the high level.* □

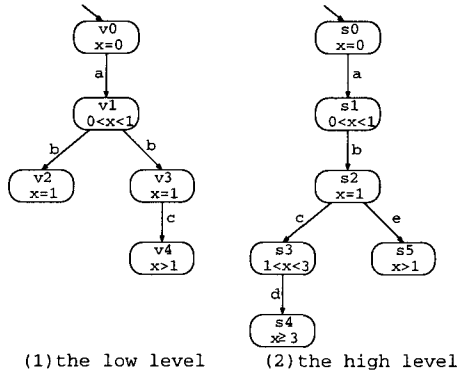


Fig. 6. Example of verifying timed simulation relations

4 Example of hierarchical design of real-time systems

4.1 Design support system

Our design support system consists of region graph generator and timed simulation verifier. We input specifications as programming language formats into region graph generator. We generate region graphs from nondeterministic timed automata by region graph generator. Region graphs are represented as adjacent lists, and are calculated by timed simulation verifier. We verify whether the low

level satisfies the high level or not by timed simulation verifier. We can verify *a safety timed simulation relation* and *a* $\exists$*-liveness timed simulation relation*, *a* $\forall$*-liveness timed simulation relation* by timed simulation verifier. If the low level satisfies the high level, we design the lower level based on the low level. If not so, we improve the low level until the low level satisfies the high level. We repeat the above steps, and finally design the lowest level. Design support system runs on SUN ULTRA(main memory 24MB), and is implemented in C language(7kstep). We show the design steps and design support system in Figure 7.

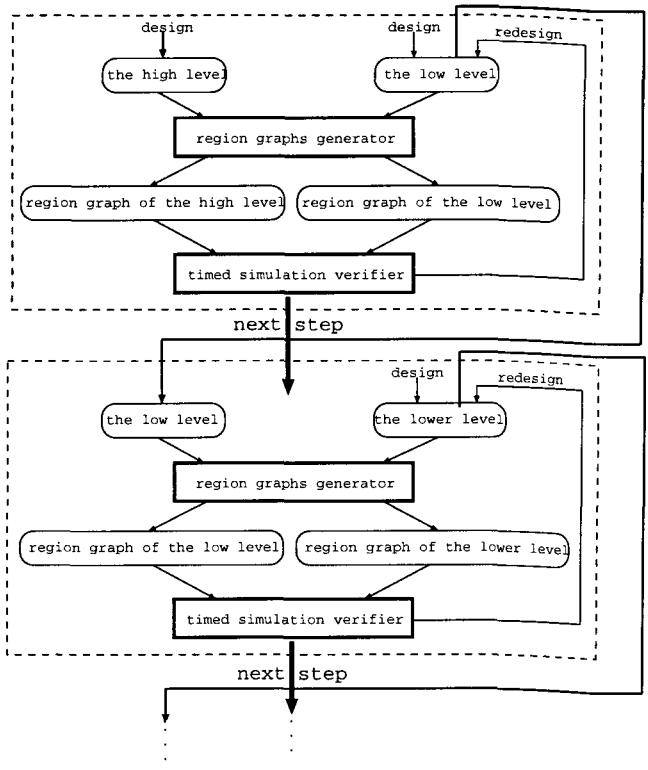


Fig. 7. Configuration of verification system

4.2 Verification example

In this paper, we show our proposed hierarchical design method effective by Ethernet CSMA/CD [IEEE 85].

Specification of CSMA/CD: Ethernet CSMA/CD is well known to the all over the world. It consists of many senders and receivers.

1. The initial state is si0. The sender stays in state si0 until it sends a message. Then, it tests the bus to see if it is ready or busy, collision detection. In the first case,it begins sending. Otherwise,it waits a some time less than or equal to 10 for a new attempt. Once the transmission is started,it will take the sender some time to complete it. Meanwhile, a collision may occur forcing the sender to wait for a while until a new attempt.

1. We specify the high level of i-sender of CAMA/CD protocol by a nondeterministic timed automaton in Figure 8(1). This means that if a sender sends data, then it repeats finite busy loops and collision detection loops, and it eventually sends data correctly. Namely, we can specify protocols with unreliable buffers by fairness constraints.
2. We specify the low level of i-sender of CAMA/CD protocol by a nondeterministic timed automaton in Figure 8(2). This means as follows : It nondeterministically behaves from an initial state. In one case, it repeats finite busy loops and collision detection loops, and it eventually sends data correctly. In another case, it can not send data correctly because of network troubles. We can easily specify this error behavior by nondeterministic timed automata.

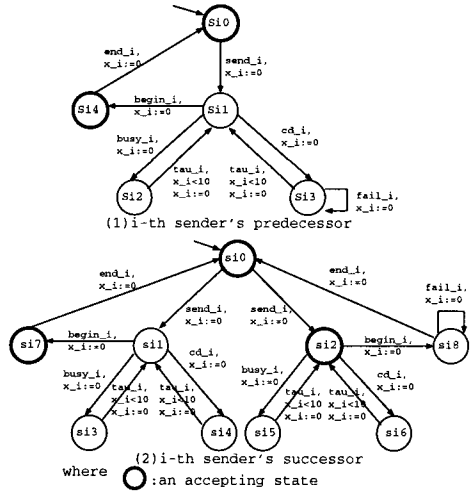


Fig. 8. Example of sender of CSMA/CD

2. At an initial state Ri0, it is ready to be in the transmission of a message. If one of the senders starts sending, the receiver sets to zero the timer yi. When yi is less than or equal to 5,the bus is sensed idle for the other sender. When the value of yi equals 5, the receiver performs an internal move labeled Ri3 to a state where it waits for a termination signal from the transmitting sender and it responds busy to the other one. When the end of the message arrives,

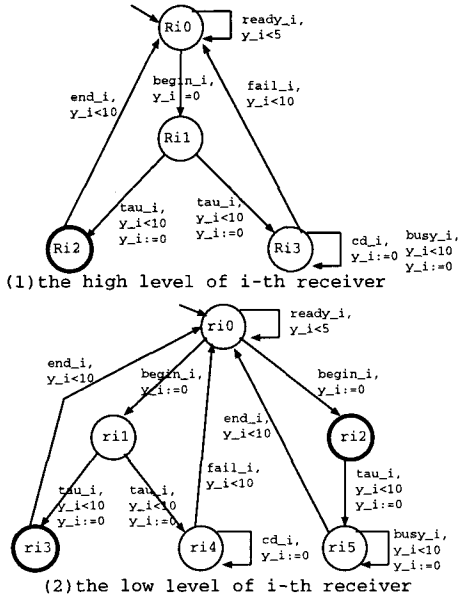


Fig. 9. Example of receiver of CSMA/CD

the receiver perform a move labeled Ri0 to a state. If it can not receive data correctly, it again receive data.

1. We specify the high level of i-receiver of CAMA/CD protocol by a nondeterministic timed automaton in Figure 9(1). This means that if a receiver receives data, then it repeats finite busy loops and collision detection loops, and it eventually sends data correctly.
2. We specify the low level of i-receivers of CAMA/CD protocol by a nondeterministic timed automaton in Figure 9(2). This means as follows : It nondeterministically behaves from an initial state. In one case, it repeats finite busy loops and collision detection loops, and it eventually receives data correctly. In another case, it can easily send data correctly without network troubles.

Verification of consistencies: Next, we verify the consistency between the low level and the high level. In this paper, communication systems consist of many senders and receivers, and they concurrently behave. Namely communication systems are represented by product automata. We verify whether there exist *a safety timed simulation relation* and *a* $\exists$*-liveness timed simulation relation*, *a* $\forall$*-liveness timed simulation relation* from the low level to the high level or not as shown in Figure 8 and 9. If we verify the consistency of communication systems by our system, there exist *a safety timed simulation relation* and *a* $\exists$*-liveness timed simulation relation.* But there does not exist *a* $\forall$*-liveness timed*

simulation relation. From these experiences, *a ∀-liveness timed simulation relation* is too strong a relation. Moreover, *a safety timed simulation relation* is not appropriate, because it does not represent fairness. We believe *a ∃-liveness timed simulation relation* is best relation between the low level and the high level. We have verified two and four, six, eight senders and receivers, and mea-

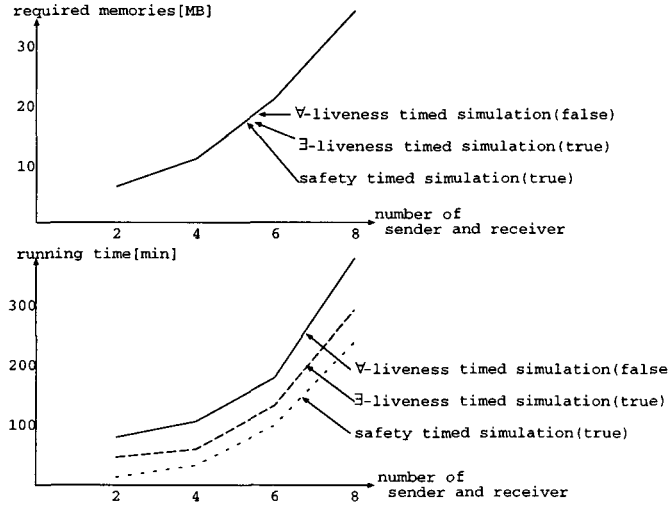


Fig. 10. running times and required memories

sured required memories and running times by time command. The results are shown in Figure 10. From these results, as the number of senders and receivers increases, required memories and running times grows exponentially. We conjecture that EXPTIME [Hopcroft 79] is also a lower bound for this problem, because the number of states of region graphs grows exponentially. But it is practically important to specify and verify real-time systems by nondeterministic timed automata, and realize hierarchical designs.

5 Conclusion

In this paper, we propose simulation relations with fairness and time. Moreover, we propose the following hierarchical design method : we describe all the specifications from requirement stage to design stage using nondeterministic timed automata, and verify the consistencies between all the specifications. Moreover, we generalize simulation relations with time and fairness as follows.

1. *safety timed simulation relation* without fairness
2. *∃-liveness timed simulation relation* satisfying fairness in some state sequence
3. *∀-liveness timed simulation relation* satisfying fairness in all state sequences

We develop the following areas in the future.

1. Symbolic verification based on BDDs(Binary Decision Diagrams) [Burch 90] for reducing required memories
2. Evaluations of required memories and running times by real problems.

References

[Alur 90] R. Alur, C. Courcoubetis, D. Dill. Model checking for real-time systems. In *Proc. of 5th LICS*, pp. 414-425, 1992.

[Alur 92] R. Alur, D. Dill. The theory of Timed automata. In *Lecture Notes in Computer Science 600*, pp.45-73, 1992.

[Burch 90] J.R. Burch, E.M. Clarke, K.L. McMillan, D. Dill, L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proc. 5th LICS*, pp.428-439, 1990.

[Cleaveland 89] R. Cleaveland, J. Parrow, B. Steffen. The concurrency workbench. In *Lecture Notes in Computer Science 407*, pp.24-37, 1989.

[Dill 91] D. Dill, A.J. Hu, H. Wong-Toi. Checking for Language Inclusion Using Simulation Preorders. In *Lecture Notes in Computer Science 575*, pp.255-265, 1991.

[Francez 86] N. Francez. Fairness. In *Texts and monographs in computer science*, Springer-verlag, P.295, 1986.

[Hopcroft 79] J.E. Hopcroft, J.D. Ullman. Introduction to automata theory, languages, computation, P.418, Addison-Wesley PUB., 1979.

[IEEE 85] IEEE Computer Society. IEEE ANSI/IEEE 802.3, ISO/DIS 8802/3. IEEE Computer Society Press, 1985.

[Milner 89] R. Milner. Communication and Concurrency. In *Prentice Hall*, P.260, 1989.

[Lynch 87] N.A. Lynch, M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. of ACM Symp. on Principles of Distributed Computing*, pp.137-151, 1987.

[Lynch 92] N.A. Lynch, H. Attiya. Using mapping to prove timing properties. In *Distributed Computing*, No.6, pp.121-139, 1992.

[Tasiran 96] S. Tasiran, R. Alur, R.P. Kurshan, R.K. Brayton. Verifying Abstracts of Timed Systems. In *LNCS 1119*, pp.546-562, 1996.

A Lightweight Approach to Formal Methods

Sten Agerholm Peter Gorm Larsen

IFAD, Forskerparken 10, DK-5230 Odense M, Denmark
E-mail: {sten,peter}@ifad.dk Web: http://www.ifad.dk

Abstract. The main current trend in applied formal methods can be characterized by the term "lightweight". Historically, formal methods have been viewed as pure alternatives to traditional development methodologies, demanding a revolutionary change in industry to adopt them. With a pragmatic, lightweight approach, the use of formal methods is complementing and improving existing development practices in a company in an evolutionary way, demonstrating more clearly the cost-effectiveness of formal methods. This paper presents our view on lightweight formal methods as a strategy for successful formal methods technology transfer to industry.

1 Introduction

Initially formal methods were promoted aggressively by idealistic academics. Formal methods were seen as *the* solution to the "software crisis" and implied the uncompromising use of mathematics and rigor for the whole development life cycle where entire software systems were developed from scratch. Hence, the development process should start with an abstract specification of the system and proceed by formal refinements and proofs of correctness towards a final implementation. From an industrial point of view this is not realistic because 1) systems are rarely developed from scratch, 2) only parts of the systems would benefit from a formal model, and 3) the general skill level in industry is not adequate to cope with the techniques for fully formal development. Many companies would be reluctant to throw away current practice and investments required by such a revolutionary approach.

Rather than focusing entirely on the ambitious certainty of correctness, the formal methods communities are starting to loosen up and find lightweight approaches to applying their technologies. For example, this new trend was documented recently by Jones [15], Jackson and Wing [14], and by Easterbrook et al. [6]. The authors use the term "light" or "lightweight" in the sense of "less-than-completely formal" or "partial" where the methods can be used to perform "partial analysis on partial specifications, without a commitment to developing and baselining complete, consistent formal specifications" [6].

With this approach, formal methods are used more as a defect detection technique in the early stages of the software development life cycle. Here, the abstraction inherently associated with formal modeling is a powerful means of reducing complexity and improving a development team's understanding of the

requirements. The motivation for the early application of formal methods is twofold. Firstly, it is commonly agreed that the expensive mistakes are made in a project's early stages [15, 11]. Secondly, current requirements engineering practice is often based on ad hoc methods and natural language specifications with no or little tool support [11, 6]. In contrast, formal notations provide a structured way of expressing requirements, which makes inspections more effective and reliable, and allow different sorts of checking to be performed by automation tools.

However, the lack of proper tools has often been claimed to be a main obstacle to the industrial take-up of formal methods. Many existing tools are merely prototypes with academic aspects such as Emacs and LaTeX. For proper take-up, tools must have industrial quality, be based on industrial platforms such as Windows, and support document generators such as Word. We have already taken this step with the IFAD VDM Tools [12, 7, 18].

The IFAD VDM Tools are particularly geared towards a lightweight approach. The key here is the focus on executable specifications which makes it easier for traditional engineers to produce and analyze formal models in a familiar testing framework. The tools also enable users to use a combination of conventional graphical techniques such as UML together with formal models described in VDM++, which is an object-oriented extension of the ISO Standard formal specification language VDM-SL [13, 10]. Further, emphasis is put on the capabilities for interaction between formally specified parts and parts of systems which are developed traditionally. Thus the IFAD VDM technology is easy to adapt by traditional software developers in an evolutionary way. The main obstacle is to teach the engineers how to choose which parts to model and how to make appropriate abstractions of these parts. In our courses we focus exactly on these aspects. It is our experience that because of the emphasis on hands-on usage of tool support with testing facilities to try out the semantics of new constructs, even engineers with no formal methods background are capable of writing appropriate VDM models after a one week course.

In this paper we provide an overview of the pragmatic, lightweight approach that we advocate to successfully introduce formal methods in an industrial setting. We focus on the requirements capturing part of the life cycle. We demonstrate how abstract models can be used as complements to graphical models. The main focus is on how such precise models can be validated and which kind of design errors can be captured with this approach. Finally, we report on some of the success stories we have had in transferring this approach to industrial end-users and give some concluding remarks on how we see the future for this trend.

2 Modeling Systems

This section first places formal modeling in a context of the software life cycle. It continues to present pointers to previous work on formal modeling using the VDM formal specification languages and on the use of these languages in conjunction with graphical modeling notations. This leads up to Sect. 3, where

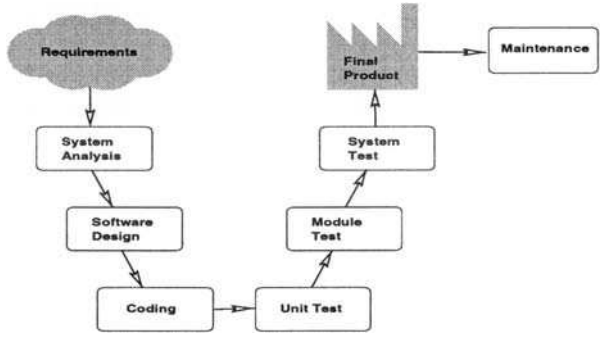


Fig. 1. Phases of the software life cycle

the same examples are used to illustrate the lightweight validation techniques facilitated by the IFAD VDM Tools. More introductory and better examples of abstract formal modeling can be found in [10].

2.1 Software Life Cycle

Many organizations use a development process which is a variant of the V life cycle model presented in Fig. 1. Starting from requirements, the development process proceeds through a number of phases until a final product is delivered. Then, the maintenance phase begins.

The traditional approach to the V life cycle model has a number of problems. A central one is that errors are discovered too late, when they are expensive to fix. This can result in overly late delivery and badly structured software due to last minute error patching. As stated in [11], the early stages of the software life cycle are especially prone to errors that can have a lasting influence on reliability, costs and safety of a system.

Essentially, the current methods used in the early stages of development are too ad hoc, as they are based mainly on natural language specifications and informal or semi-formal diagrammatic notations. These make inspections woolly and not so useful in locating errors, in contrast to, for example, code inspections [15]. Moreover, such descriptions do not allow automation tools, so the reviewing process becomes a manual process that is subject to human limitations and costly to repeat when requirements change.

Formal modeling is viewed as a way of improving the system analysis phase. It can help to achieve a better understanding of requirements and to reduce risk in the development process by finding defects early. The following aspects of formal modeling are considered important:

Abstraction: The abstraction supported by formal specification languages can help to cope with the complexity of a system to be developed. Abstract models are built by focusing on a specific purpose of the analysis and including

only the relevant aspects of the system in the model being built. Russell said: "Abstraction, difficult as it is, is the source of all practical power."

Precise modeling: Formal notations can help to present the requirements in a structured manner, which makes reviewing and inspection easier and therefore useful in locating errors.

Tools: Formal notations allow tools to be developed for automating the analysis and inspection. Examples range from basic syntax and type checkers to interpreters for executing specifications and automated theorem provers.

Section 3 discusses the use of tools to validate formal models.

2.2 Modeling in VDM-SL

A formal specification language such as VDM-SL supports the development of abstract descriptions of data and functional behavior of a system. Its data type constructors for sets, sequences and mappings, and their associated constructions such as comprehensions and first-order quantifications, let the engineer concentrate on "what the system must do, not how it should do it" [16]. Moreover, the engineer can document desired properties explicitly in the model through the use of invariants and pre- and post-conditions.

Metro Door Management Example In [3], a VDM-SL model of a metro door management system was presented. The main safety requirement of such a system is that the doors must not be open at any time while the train is moving, which is essential for the safety of the passengers.

By focusing the purpose of the analysis on ensuring that a model satisfies this requirement, we can reduce the complexity of a metro system drastically. The central components of the system are the train and the doors, modeled as a record type definition in VDM-SL:

```
Metro :: train: Train
         doors: Doors;
```

The only thing we need to know about the train and the doors is their state, modeled as the following enumeration type definitions:

```
Train = <stopped> | <moving>;

Doors = <open> | <closed> | <halfopen>;
```

(This model is a cut-down version of the one presented in [3].)

The safety requirement can be formalized and documented in the model directly, by introducing an invariant on the metro record type above:

```
Metro :: train: Train
         doors: Doors
inv metro ==
  not(metro.train = <moving> and metro.doors <> <closed>);
```

As we shall see in Sect. 3, such a property is checked automatically in executing the specification.

The model provides a very state diagram oriented view on the metro system. In fact, the model was inspired by some OMT state diagrams for the train and the doors, and the main functions of the VDM-SL model formalize the transitions defined in these diagrams (see Sect. 2.3). For example, the doors transition function is defined along these lines:

```
DoorsTransition: Doors * Button * Train -> Doors
DoorsTransition(doors,butt,train) ==
  cases doors:
    <open>     -> if butt = <do_close>
                  then <halfopen>
                  else <open>,
    <closed>   -> if butt = <do_open> and train = <stopped>
                  then <open>
                  else <closed>,
    <halfopen> -> ...
  end;
```

The VDM-SL model contributed a more rigorous view than OMT, e.g. through the use of the invariant and an executable model. Surprisingly, we were able to break the invariant in validation (see Sect. 3). VDM-SL would support more abstract, less state oriented models of the metro system.

SAFER Example In [2], we presented a VDM-SL model of the SAFER system, which is a small, lightweight propulsive backpack module designed to provide self-rescue capabilities to a NASA space crewmember. It is used in emergency situations during an extravehicular activity if a crewmember becomes separated from a space station or a space shuttle orbiter docked to a space station.

The SAFER system was described in the NASA Formal Methods Guidebook [19], which included both the informally stated requirements and a PVS model of the thruster selection functionality. In response to translation and rotation commands issued by a crewmember via a hand controller module, and in response to rotation commands issued by an automatic attitude hold capability (AAH) which a crewmember can activate to detumble himself, the SAFER software must select a certain number of gaseous nitrogen (GN_2) thrusters to fire, out of possible 24. We essentially translated the PVS model to VDM-SL in order to show how validation by testing could be viewed as a complement to verification by proof used in the guidebook (see Sect. 3). The purpose of the model is to check that the thruster selection functionality works properly, and thus, for example, the actual calculation of the AAH output is outside the scope of the model (following the Guidebook).

As the SAFER example is too big to be presented at length in this paper, we shall only discuss one important aspect of our model related to two main safety requirements of the SAFER system listed in [19].

Property (41): The avionics software shall provide accelerations with a maximum of four simultaneous thruster firing commands.

Property (E1): Thruster firing consistency: No two selected thrusters should oppose each other, i.e., have canceling thrust with respect to the center of mass.

A central operation in this respect is the control cycle, which reads the crewmember commands and the AAH output command as inputs, and calculates the thruster settings. By defining a post-condition on the control cycle we are able to document these properties in our model:

```
ControlCycle: HCM'HandGrip * AUX'RotCommand * ... ==>
              set of TS'ThrusterName
ControlCycle(raw_grip,aah_cmd,...) ==
  let grip_cmd  = HCM'GripCommand(raw_grip,...),
      thrusters = TS'SelectedThrusters(grip_cmd,aah_cmd,...)
  in
     (AAH'Transition(grip_cmd,clock,...);
      clock := clock + 1;
      return thrusters)
post card RESULT <= 4 and ThrusterConsistency(RESULT)
```

The specification is modular, where back quote is used to prefix with a module's name. First the raw hand grip command is converted to a combined translation and rotation command and then the thruster settings are calculated. The control cycle maintains a clock and a state in the automatic attitude hold module, before the set of thruster settings is returned. The post-condition uses an auxiliary function for thruster consistency, defined as follows:

```
ThrusterConsistency: set of TS'ThrusterName -> bool
ThrusterConsistency(thrusters) ==
  not ({<B1>,<F1>} subset thrusters) and
  not ({<B2>,<F2>} subset thrusters) and
  not ({<B3>,<F3>} subset thrusters) and
  not ({<B4>,<F4>} subset thrusters) and ...
```

Section 3 shows how validation is used to achieve a high confidence that the model satisfies the post-condition.

2.3 Combining Graphical and Formal Modeling

Despite all benefits, formal models can be complicated to organize and structure during development. Often diagrammatic views, e.g. in object-oriented modeling, can help to master the complexity of software systems. Graphical notations can give an abstract, structural overview of a model, which can significantly aid in a developer's understanding of the model. Hence, we believe that graphical notations such as UML are particularly well-suited for making the first sketches of an object-oriented model of a software system, presenting and visualizing the

main components of a system and the relations between them in a high level way. Moreover, graphical notations are becoming widely used in industry and so can be an entry point for formal methods application in industrial practice.

Metro Example In the metro example presented above, OMT was used to analyze the requirements prior to formally specifying a model of the system. As noted in [17], this approach can help to enhance the initial formal specifications and reduce the effort required to produce them, but we also felt that this made the focus of the model more low-level as it became very state oriented. The example also showed an instance of the misunderstanding of a graphical model, because OMT makes an implicit assumption that events cannot happen simultaneously. The translation to VDM-SL did not respect this semantics, which made it possible to violate the main safety requirement (see Sect. 3).

SAFER Example In a recent paper [5], the SAFER example has been reworked in an alternative, object-oriented development rather than the modular approach in [19,2]. This makes use of graphical modeling using UML in conjunction with formal modeling using VDM++, an object-oriented extension of VDM-SL. The overall object-oriented aspects of SAFER are first modeled in a UML object model, which is translated to VDM++ for further analysis by adding concise type information and functional behavior to the model. The transition forward and backward between the two notations is supported by the Rose-VDM++ Link [4]. This provides round trip engineering capabilities to the user by integrating Rational Rose and the IFAD VDM Tools. In this way, the complementary benefits of graphical and formal modeling can be exploited effectively, as visualization becomes a direct and key element of the modeling process, where the graphical notation presents a structural overview and the formal notation enables a rigorous analysis by automation tools.

3 Validating Models

Though experience shows that valuable insight can be gained already by writing a formal model [16,6], formal modeling alone does not provide a clear enough argument for industry's adoption of formal methods. It has to be accompanied by professional, quality tools, not to mention training and consultancy, which are targeted at the current skills of traditional engineers. By incorporating formal methods into the software practice in small deltas [9], the technology transfer process is likely to have a more significant effect.

Therefore we propose the use of testing techniques to validate formal models. In industry, these techniques are already commonly used and known to be effective tools in finding defects. Hence, there is a much clearer cost-benefit argument of this approach than with traditional formal methods, since it is commonly agreed that it is desirable and cost-saving to find errors early in the life cycle.

Moreover, the test cases applied to the formal model can be reused on the final implementation, providing a strong argument for conformance of the code

with respect to the requirements. In this way, the formal model acts as an oracle against which the implementation is tested in an automated way, complementing traditional code inspections and perhaps reducing the effort and cost required to conduct these.

3.1 Executing Specifications

Executable specifications are a prerequisite for applying validation using testing techniques. Another essential thing is a tool for performing the execution, as clearly hand-execution is not feasible.

A large subset of VDM (i.e. VDM-SL and VDM++) is executable and supported by an integrated interpreter and debugger of the IFAD VDM Tools. The interpreter can be used interactively by feeding it with expressions or script files, but it can also be used in batch mode for evaluating large test suites.

Test Expressions The interpreter can read and execute any VDM expression in the executable subset. These include trivial expressions like `5+3` and function calls, and expressions such as the map comprehension expression

```
{mk_(a,b,c,d,m) |->
   SAFER`ControlCycle(mk_HCM`HandGrip(a,b,c,d),...,m,...) |
 a,b,c,d in set {<zero>,<pos>,<neg>}, m in set {<tran>,<rot>}}
```

which in a compact way provides a kind of test case generation expression [2]. The above expression executes the SAFER control cycle 162 times and returns a mapping from inputs to the corresponding outputs.

Test Statistics and Test Coverage During execution the interpreter collects a number of test statistics and it can be asked to show test coverage information in Word (or LaTeX) documents, using multiple fonts or coloring to distinguish the executed and non-executed parts. Such information can be an aid in designing test cases, but can also aid in locating errors in a specification. This was illustrated in [5], where the test statistics said a certain function was executed only 6 times with a coverage of 81%, while we expected 81 executions with 100% coverage. The test coverage coloring led us on the right track for detecting the error, and the correction was easy.

In the SAFER example [2], we used the test coverage coloring to increase our confidence that certain non-applicable cases stated in large tables of the requirements document were indeed never executed. Using a map comprehension as above, the control cycle was executed 8748 times, yielding a coverage of only 72% of one of the tables, though executed in every cycle. The tables were represented as functions in VDM-SL using cases expressions of the form:

```
LRUDThruster(a,b,c) ==
cases mk_(a,b,c):
  mk_(<neg>,<neg>,<zero>) -> mk_({},{}),
```

```
    mk_(<neg>,<zero>,<zero>) -> mk_({<L1R>,<L3R>},{<L1F>,<L3F>}),
    mk_(<pos>,<zero>,<pos>) -> mk_({<R2R>},{<R2F>,<R4F>}),
    ...
end;
```

Here, the test coverage facility shows the non-executed parts in boxes, and these correspond exactly to the non-applicable cases.

3.2 Consistency Checking during Execution

While specifications are executed, the interpreter checks consistency of assumptions and requirements documented in invariants and pre- and post-conditions. This can be an aid in finding bugs or gaining confidence in models when a large enough set of test cases is executed successfully.

Metro Example: Invariant Checking In the metro model, we used an invariant to state the main safety requirement of the system, saying that the doors should not be open while the train is moving. In several executions, the invariant was checked and found to be true. However, by defining and executing a certain scenario it was possible to break the invariant, reported as a runtime error by the interpreter. This happens in the state where the doors are closed and the train is stopped, when we ask the doors to open and the train to accelerate at the same time [3]. In this case, the doors will start to open and the train will start to move, which violates the invariant on the metro type.

The same situation cannot arise in the OMT state diagrams from which the VDM-SL model was strongly inspired. OMT operates with an idealized model where events can happen infinitely close, but not simultaneously. This implicit assumption was forgotten while translating the model to VDM-SL. This would yield an equivalent problem if the final code was produced from OMT directly.

SAFER Example: Post-condition Checking In the SAFER model, we used a post-condition on the control cycle to state the properties (41) and (E1), saying that at most four thrusters should be selected at the same time and that no two selected thrusters should oppose each other. This means that in all executions of the control cycle these properties are checked automatically. As mentioned above, we formulated a compact map comprehension expression yielding 8748 executions of the control cycle. These gave 189 different thruster settings as output, which all satisfied the post-condition. The large number of test executions gave us confidence in both the stability and conformance of our model with respect to the above mentioned requirements.

In the NASA Guidebook [19], verification by interactive theorem proving with PVS [21] was used to illustrate the analysis of property (41). The proof required considerably more skill to formulate and conduct than the automated analysis with execution described above. However, the level of confidence obtained in the model as a result of verification by proof is higher than with validation by testing.

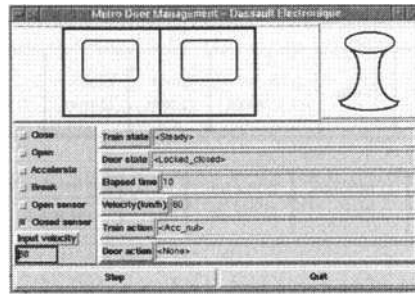


Fig. 2. Graphical front-end for metro door management model

Therefore, we propose to apply validation by testing first to achieve a certain level of confidence in a model, and then proceed by verification if demanded or beneficial. Moreover, we believe that engineers can use validation by testing cost-effectively early in their formal methods careers, but it is unlikely that the same holds for verification.

3.3 Integrating External Components

In an industrial setting, systems are rarely developed from scratch and will often consist of parts where formal modeling is not useful, e.g. a graphical user interface or database handling. In order to integrate formal modeling with the development process, it is therefore essential to allow the formal model to include external components, for example, to facilitate testing of the model in conjunction with already developed parts of a system. The IFAD VDM Tools provide such functionality through a Dynamic Link facility which allows a user to execute code together with a VDM model.

In addition to supporting the technology transfer process, the integration of external components into a model can support the assessment of a formal model by a customer or staff member not trained in VDM. By developing a graphical front-end and linking this with a model, a customer or colleague can "inspect" a model in an interactive way without reading the VDM, and thus get a better understanding of the specifier's view of the requirements. This use of formal modeling and the Dynamic Link facility enables rapid prototyping.

Metro Example Front-end In the metro case study, a graphical front-end was developed quickly. Though rather simple, it had great value for demonstration purposes, but also for the engineers themselves in order to ease the testing process. Once appropriate test scenarios were defined they were easy to enter through the graphical front-end.

Figure 2 presents one view of the metro door management front-end. The top part shows the metro doors and a bell which is used to warn passengers before doors start to close (this aspect was excluded from our description in Sect. 2.2).

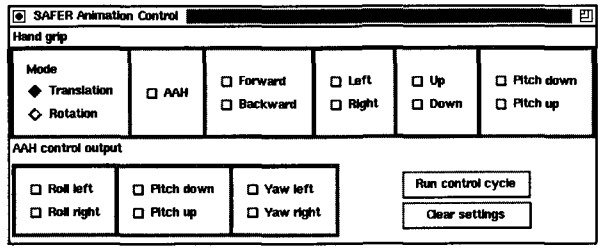


Fig. 3. Graphical model of the SAFER hand controller

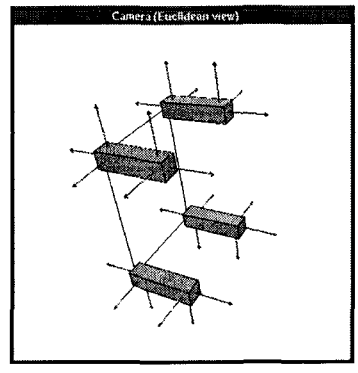


Fig. 4. Graphical model of the SAFER backpack

The left part shows the input to the VDM model, which are buttons operated by the driver, door sensors and the speed of the train. The right part shows output from the model and various status information. Hence, the front-end provides a visualization of the VDM model which a customer or colleague can immediately understand and interact with.

SAFER Example Front-end In the SAFER example, we developed a graphical model of the hand controller unit used by the crewmember to issue commands to the thruster control software (see Fig. 3). The front-end also includes a panel for inputting the rotation command of the automatic attitude hold (AAH), as its calculation was abstracted away from the VDM model.

Moreover, we developed a graphical model of the backpack module in order to visualize the thruster settings calculated by the VDM model. For illustration, Figure 4 shows all possible 24 thruster settings.

These figures could naturally be enhanced significantly, but the purpose here is just to illustrate the feasibility of the approach. For example, in testing the SAFER model we can simply enter commands via the hand controller front-end

rather than by writing complex input arguments for the SAFER control cycle. These are generated automatically by the code underneath the front-end.

The graphical front-ends were a major help in validating a number of properties mentioned in the requirements document for SAFER [19]. During this analysis process we found some interesting "unclarities" in the requirements by using the front-ends in a structured way to enter a number of test scenarios and immediately check the visualized results (see [2]).

4 Industrial Application Success Stories

This section provides some pointers to a few projects where the lightweight approach to formal methods has worked successfully as a strategy for formal methods technology transfer to industry.

4.1 Trusted Gateway by BASE

At British Aerospace (Systems and Equipment) a controlled experiment was carried out measuring the effects of introducing formal modeling of a small security-critical component known as the trusted gateway [16,8]. The trusted gateway acts as a filter between a high-security system and less secure systems, ensuring that messages of high secrecy are not passed to systems deemed too insecure to handle them.

In this project a number of comparisons was made between two separate developments of the trusted gateway, one development using conventional practice, the other using VDM-SL in addition to this when considered helpful. The conventional techniques included the graphical notation SA in the Ward and Mellor dialect, without special support for the combination with VDM-SL.

The main conclusion from this project was that the use of formal modeling and VDM Tools in the early phases was beneficial and not more expensive than a conventional development. Moreover, the conventional development failed to handle an exceptional case which was clarified by the formal development in the system analysis phase. This error was not found until after the acceptance testing, and the patch for this made the software developed conventionally fourteen times slower than the software developed using VDM.

Based on the experience, the VDM technology was recommended in future projects at BASE. We know of at least three more projects that have been carried out using the VDM technology, but they have not needed our assistance in any of these projects. This is a clear indication that the technology transfer process has worked successfully.

4.2 Metro Door Management by Dassault Electronique

Dassault Electronique (D.E.) is a major French provider of safety-critical systems for terrestrial transportation, space, and military and commercial avionics. In these markets, the use of formal methods is often mandatory or encouraged,

for example, the French RATP (Régie Autonome des Transports Parisiens) mandates the use of the B method [1] for the development of safety-critical software for the Paris Metro.

While some French authorities encourage the use of the B method, D.E. find that B does not meet their needs early in the software development process, when requirements are often changing and may need to be clarified and agreed upon by customers. They evaluated the IFAD VDM Tools in a case study focusing on the door management system of a metro. The results were presented in [3], and have been used for illustration throughout this paper.

The biggest surprise to Dassault Electronique was that they were able to master the VDM technology so quickly. After only three days of training in a one-week training course, they were able to both write and do the initial analysis of their first model of the door management system, based on the informal and semi-formal (OMT) requirements document written in advance. They believe that the support for animation, testing and rapid prototyping provided by VDM Tools is very valuable in the early stages of development. For example, this helps to clarify and better understand requirements and to present the developer's view of the requirements to the customer. By using the VDM technology prior to a development in B, they estimate to achieve a total cost reduction of 15–30% compared to the costs of using B for the entire development.

4.3 Applications in the Non-critical Domain

Traditionally, formal methods have been promoted mainly for safety-, security-, or mission-critical systems, as a tool to ensure the "correctness" of such systems. However, in particular, lightweight formal methods can be used in a much wider spectrum of applications. It is therefore very encouraging that we are starting to see applications of the VDM technology outside the critical sector. Two of these are described briefly below.

Banknote Processing by GAO The Gesellschaft für Automation und Organisation mbH (GAO) in Germany develops banknote processing systems. Such machines are very complex with embedded computer systems which must be extremely configurable with respect to both types of sensors and bank-notes to be analyzed. The volume of the details and the number of potential error cases involved in its design proved to be too extensive to be clearly expressed in an informal notation. By formulating the abstract model in VDM, and executing realistic test scenarios with the IFAD VDM Tools interpreter, the software engineers were able to gain confidence that their design was watertight. Only one week of training was necessary to enable them to use formal modeling beneficially in an industrial context.

Sales Configuration by Baan Front Office Systems At Baan Front Office Systems they develop a general tool – SalesPLUS – for the configuration of services and products. For the next generation of the salesPLUS configurators they

are developing a new declarative programming language jointly with an American company. The language is currently undergoing a language analysis phase using VDM in the lightweight fashion described in this paper. A large group of people from both Denmark and USA have been trained in this technology and they feel that this approach has had a number of advantages:

- Common basis for precise discussions.
- Improved cooperation over long distances.
- Testing of VDM models and early construction of test environments.
- Faster route to a prototype.

This project is of vital importance for Baan Front Office Systems and the use of VDM is placed right in the heart of the development process.

5 Conclusions

In this paper we have presented a lightweight approach to formal methods with examples of its use and pointers to a few of the industrial applications. Through formal modeling in an executable notation, the user gets access to validation facilities based on the testing techniques that he is already comfortable with. Hence, the introduction of the IFAD VDM technology in a company is a small, evolutionary change of existing practice, but a big and cost-effective improvement. Training efforts are usually limited to a one-week course. In our opinion the technology transfer is only successful when traditional engineers use the technology rather than formal methods experts as suggested in [6].

The application of our pragmatic, lightweight approach is focused on the early stages of the development process. At this point, the use of abstraction is important to cope with the complexity of systems to better understand the requirements. Moreover, the use of techniques and automation tools that are effective for locating defects is essential at these stages as the expensive mistakes are known to be made here. By integrating external components such as graphical front-ends with the execution of formal models, it becomes possible for a customer or non-trained staff member to assess and interact with a VDM model in a direct way.

Once the requirements are properly analyzed and understood, experience shows that it is relatively easy to implement the system using conventional techniques. Moreover, automatic code generation as supported by the IFAD VDM Tools can also be extremely valuable. However, the trade-off usually is that models need to be more low-level to allow code generation than desirable in requirements analysis.

In addition to executable models, the use of graphical modeling in conjunction with formal modeling is an important contribution to the lightweight approach of the IFAD VDM technology. Through round trip engineering capabilities between UML and the object-oriented formal specification language VDM++, the user gets direct and immediate access to the complementary benefits of graphical and formal modeling. The graphical, diagrammatic view is used

for the structural visualization of high level aspects of a model, while the formal, textual view is used for the formalization of processing details of a model, enabling the analysis of behavioral aspects by automation tools. As graphical modeling is already widely used in industry, such practical combinations can be viewed as simple, cost-effective improvements of existing practice, and may open up many new possibilities of industrial application of formal methods.

The lightweight approach presented in this paper is viewed as a first step improvement of the software development process in industry. At a later stage we also believe that more rigorous approaches including verification may be viable for critical applications, but many enhancements of the existing technology are needed before this will be the case. We are entering a major new Esprit funded project on verification called PROSPER [20], where the industrialization of verification technology will be the long term goal.

We believe that the lightweight trend in applied formal methods continues to provide a really strong strategy for formal methods technology transfer to industry. We are currently experiencing at IFAD that software organizations in the non-critical domains are starting to use the VDM technology. In the Autumn 1998, we already know that Matra, Dassault Aviation, CISI and Bruhn NewTech will start to use VDM in the lightweight fashion in new projects measuring the effect of this. So far the results from, for example, Baan Front Office Systems and GAO look very promising. We expect the main expansion of our users to come from the non-critical domains where our pragmatic approach will be adopted because of cost and quality arguments.

Acknowledgments

We would like to thank Bernhard Aichernig for comments on a draft of the paper.

References

1. J.-R. Abrial. *The B Book – Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. S. Agerholm and P.G. Larsen. Modeling and Validating SAFER in VDM-SL. In *Fourth NASA Langley Formal Methods Workshop.* NASA, September 1997. NASA Conference Publication 3356. Available at http://atb-www.larc.nasa.gov/Lfm97/.
3. S. Agerholm, P.-J. Lecoeur, and E. Reichert. Formal Specification and Validation at Work: A Case Study using VDM-SL. In *Proceedings of Second Workshop on Formal Methods in Software Practice.* ACM, Marts 1998.
4. S. Agerholm and O.S. Pedersen. Enhanced UML Analysis with Formal Modeling and Validation. Draft, Marts 1998.
5. S. Agerholm and W. Schafer. Analyzing SAFER using UML and VDM++. Draft, Marts 1998.
6. S. Easterbrook, R.R. Lutz, R. Covington, J.C. Kelly, Y. Ampo, and D. Hamilton. Experiences Using Lightweight Formal Methods for Requirements Modeling. *IEEE Transactions on Software Engineering*, 24(1):1–11, January 1998.

7. R. Elmstrøm, P.G. Larsen, and P.B. Lassen. The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. *ACM Sigplan Notices*, 29(9):77–80, September 1994.
8. J.S. Fitzgerald. ESSI Project ConForm: Home Page. WWW at URL http://www.csr.ncl.ac.uk/projects/ConForm.html, 1994.
9. J.S. Fitzgerald and P.G. Larsen. Formal Specification Techniques in the Commercial Development Process. In M. Wirsing, editor, *Position Papers from the Workshop on Formal Methods Application in Software Engineering Practice, International Conference on Software Engineering (ICSE-17), Seattle*, April 1995. ftp://ftp.ifad.dk/pub/papers/icse.ps.gz.
10. J.S. Fitzgerald and P.G. Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development.* Cambridge University Press, 1998.
11. D. Hamilton, R. Covington, and J.C. Kelly. Experience in Applying Formal Methods to the Analysis of Software and System Requirements. In *Workshop on Industrial-Strength Formal Specification Techniques*, pages 30–43. IEEE Computer Society Press, April 1995.
12. IFAD World Wide Web. `http://www.ifad.dk`.
13. International Standard, ISO/IEC 13817-1. Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language, December 1996.
14. D. Jackson and J. Wing. Lightweight Formal Methods. *IEEE Computer*, 29(4):22–23, April 1996.
15. C.B. Jones. A Rigorous Approach to Formal Methods. *IEEE Computer*, 29(4):20–21, April 1996.
16. P.G. Larsen, J.S. Fitzgerald, and T. Brookes. Applying Formal Specification in Industry. *IEEE Software*, 13(3):48–56, May 1996.
17. R.R. Lutz. Reuse of a Formal Model for Requirements Validation. In *Fourth NASA Langley Formal Methods Workshop.* NASA, September 1997. NASA Conference Publication 3356. Available at http://atb-www.larc.nasa.gov/Lfm97/.
18. P. Mukherjee. Computer-aided Validation of Formal Specifications. *Software Engineering Journal*, pages 133–140, July 1995.
19. NASA. Formal Methods, Specification and Verification Guidebook for Verification of Software and Computer Systems. Vol 2: A Practitioner's Companion. Technical Report NASA-GB-001-97, Washington, DC 20546, USA, May 1997. Available from http://eis.jpl.nasa.gov/quality/Formal_Methods/.
20. PROSPER World Wide Web. `http://www.dcs.gla.ac.uk/prosper`.
21. PVS World Wide Web. `http://www.csl.sri.com/pvs.html`.

An Open Environment for the Integration of Heterogeneous Modelling Techniques and Tools

Robert Büssow[1], Wolfgang Grieskamp[1],
Winfried Heicking[2], and Stephan Herrmann[1]

[1] Technische Universität Berlin, Institut für Kommunikations- und Softwaretechnik, FR 5-6, Franklinstr. 28/29, D-10587 Berlin. {buessow, wg, stephan}@cs.tu-berlin.de
[2] Fraunhofer-Institut für Software- und Systemtechnik ISST
Mollstr. 1, D-10178 Berlin. Winfried.Heicking@isst.fhg.de

1 Introduction

Even though the development of formal methods makes steady progress with respect to techniques and tool support, their acceptance in industry is still relatively low. One reason for this probably stems from the fact that most formal approaches require users to forget about their conventional techniques and tools, and to relearn in a new environment. What makes this switch even harder to accept: due to their research character, tools for formal methods can usually not compete w.r.t. convenience, stability, and degree of integration to conventional, state-of-the-art commercial tools.

Applying modern repository-based integration techniques to tools for formal methods can overcome some of these drawbacks. Moreover, an *incremental migration*, in which users can continue to apply their established conventional techniques and tools, adding formal support in degrees as desired, seems to be promising for bringing formal methods to practice.

A framework based on the idea of combining established modeling techniques and tools with formal ones has been developed in the context of the ESPRESS project[1]: among others, Statecharts [8], a widely used graphical notation for reactive behavior, are combined with the formal notation Z [20], and with temporal logic [3, 2]. The resulting combined notation is supported by a specification methodology [7] and, last but not least, by an open and scalable tool integration environment – the design principles of which will be sketched in this paper.

The applicability of our approach has been demonstrated in two reference case-studies of the ESPRESS project that where provided by the industrial partners. One case-study is the specification of a traffic light system [1], the second [7] is the specification of an intelligent cruise control.

Objectives. Our goal is to offer an environment for editing, browsing, and analyzing integrated specifications, assembled from heterogeneous formalisms – such

[1] The ESPRESS project is a cooperation of industry and research institutes funded by the German Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie.

as Statecharts, Z, temporal logic, event diagrams, etc. These formalisms are processed by different techniques, such as deduction-based analysis, model checking, systematic testing, simulation, and more. Established CASE tools (such as Statemate [9]) shall be used in combination with the strength of existing tools for formal methods (such as the deduction system Isabelle [17] or the model checker FDR [6]), and specialized tools newly developed in the application context of ESPRESS. Integration has to regard the following aspects:

- *Data Integration.* The various tools that are to be combined within the tool environment usually have their own, proprietary data format. The tool environment should provide uniform data formats, into which this data can be mapped, and should store these formats in a common repository.
- *Control Integration.* Tools may require certain data formats that are established by other tools (for example, a compiler expects a type-checker to be successfully run). The automatic control of *tool-chains* is therefore an important objective.
- *Presentation Integration.* As far as possible, interaction with the diverse tools should be standardized and follow modern design principles for GUIs. Yet, it is important to recognize that this goal cannot fully be accomplished with our approach: tools such as Statemate have their own closed world, and cannot smoothly be embedded in broader environments. It is particular crucial to deal with these "rough edges" of integration in our environment.

Outline of the Approach. We support a notion of a *compound heterogeneous specification document* that, in the form of a hypertext document, comprises the aggregation of models originated by different sources (cf. Sec. 2).

We define several *uniform data formats*, which may or may not be supported by each tool integrated in our framework – depending on the degree of its integration (cf. Sec. 3). Examples for formats are LaTeX, Postscript, GIF and abstract syntax.

We define interfaces of *adaptors* for tools to be integrated in our environment. A *model source adaptor* integrates an external modeling tool such as Statemate, by implementing the translation of proprietary data formats into the uniform ones. A *function adaptor* maps a set of formats into a refined set of formats, typically representing an analysis or transformation tool. The activation of both kinds of adaptors is automatically controlled by the environment. See Sec. 4.

Technically, the data formats and the adaptors for tool integration are provided in the language Java. For the definition of abstract syntax, we use the Pizza language [16], a superset of Java which provides algebraic data types and more. The code produced by the Pizza compiler can be accessed from Java programs as well. Due to its portability, powerful library support, and inter-language facilities, Java/Pizza is an ideal choice for providing interfaces to our environment. The implementation of persistency and tool control bases on the systems H-PCTE [12] and PIROL [13], which will be discussed in Sec. 5.

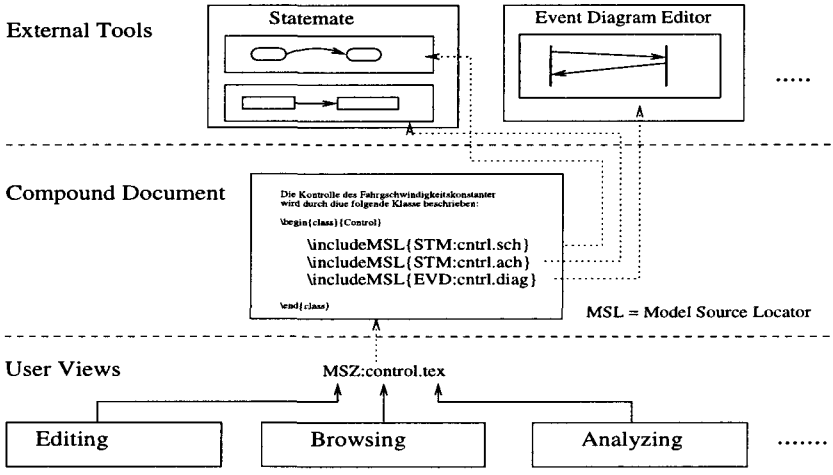


Fig. 1. Compound Heterogeneous Specification Documents

2 Compound Specification Documents

The ESPRESS methodology advocates the combination of different, specialized modelling techniques. By now, these are Statecharts, a version of dataflow-diagrams (Statemate activity charts), Z, and an extension of Z by temporal logic. These and possible other formalisms are combined into a Z-based notation, called μSZ [2], which is the basis for the *semantic integration* of formalisms for reactive systems that adopt the synchronous approach (such as Statecharts and temporal logic). The discussion of μSZ is out of the scope of this paper; we just mention that μSZ provides in particular means for structuring and modularization, which allows us to cluster the different modeling techniques and provide them with uniform interfaces.

In spite of the semantical integration, different sub-models of specifications are still treated by different tools. The specification document is a *heterogeneous compound document* given in a markup-language (currently LaTeX); the references to aggregated sub-models (and also to the overall model) are described by so-called *model source locators* (MSL), a concept comparable to URLs. An MSL identifies the tool that is to provide the (sub-)model and a tool specific identifier for the desired portion. This is illustrated in Fig. 1.

The major difference between the concept of URLs and of our MSLs is that the latter are interpreted in different ways, depending on the operation to be performed on a compound document. For example, if a document is browsed, an MSL is resolved to encapsulated postscript. If it is analyzed, it is resolved to an abstract syntax term.

Fig. 2 illustrates by a screen-shot the editing of a compound document in our environment. The compound document itself is edited within the XEmacs editor, which has been extended by visualization support for the Z notation (Z as well as

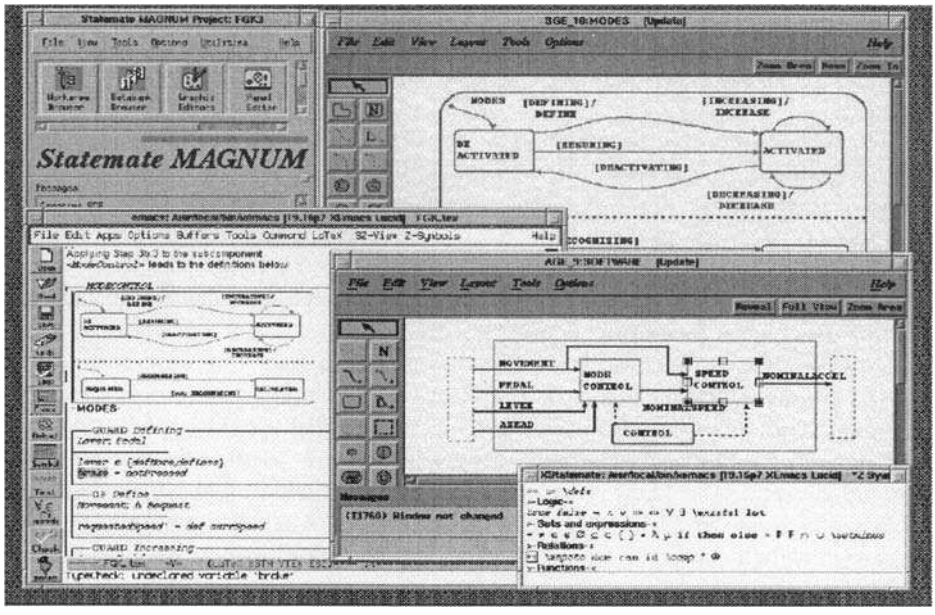


Fig. 2. Editing a Compound Document with Statemate and XEmacs

arbitrary documentation may be literally entered into a compound document). MSLs referring to Statecharts and activity charts of the Statemate tool are resolved to bitmaps which are inserted into the XEmacs editing buffer. When the user changes the charts in the Statemate tool, the bitmaps under XEmacs will be updated automatically. In the sequel, we will mainly concentrate on the technical concepts which make editing, browsing and analyzing of compound documents possible.

3 Data Integration

In our environment a collection of tools, textual and graphical editors, simulators, analysis tools, code generators, etc., work together on an heterogeneous compound specification document. To make this possible, a uniform data representation is indispensable.

To this end, we define a set of standardized *data formats*. Each MSL is dynamically associated with a set of such formats, which represent different views of the contents of the MSL. Establishing a particular format for an MSL may require invoking tool adaptors as discussed in the next section, and may generally fail. For example, establishing the format of type-checked abstract syntax requires (at least) to run a type checker, which may reject establishing this format because of type-errors.

Formats are attributed with a *qualifier*, which describe the role the format plays for the specification (such as "the test-cases derived from this specification"). This is illustrated by the tool chain in Fig. 3, as explained in the next section.

The most elaborated data format we provide is abstract syntax (called ZIRP, the Z Intermediate RePresentation format). This format is a compact representation of the full Z language, together with the features of our integration formalism, $\mu\mathcal{SZ}$. In addition to a qualifier, a ZIRP format is parameterized by a *state*. States signify e.g. whether the abstract syntax is type-checked ("certified"), and may imply that the abstract syntax is annotated with additional information, such as e.g. type information.

In general, it is possible for a tool developer to add new data formats to the environment, or to refine existing ones (in particular, to refine ZIRP). For example, if an extended type checker for Z is added to the environment, it may define new kinds of type-annotations, which are asserted by a state, say, "extended-certified". These new annotations may then be used by other new tools, whereas existing tools don't need to see them. As another example, a proof-checker may define a totally private format "proof-script", which, associated with an MSL, may be used to replay proofs on a specification with this particular checker.

Even though, it is possible to add any data format to our environment, *semantic* integration can only be provided if there exists a mapping into $\mu\mathcal{SZ}$. Thus the semantic integration of a new modeling technique in our framework relies on the translation of proprietary data formats into the abstract syntax of $\mu\mathcal{SZ}$, ZIRP. Assuming the individual tools behave correctly, the correctness of the integration depends on the foundation of the given translation – which must be approved w.r.t. the semantics of the source formalism and of $\mu\mathcal{SZ}$ – and the correctness of its implementation. Yet, that we have chosen Z/$\mu\mathcal{SZ}$ as our semantic reference language, is not fundamental to our approach; it could have been other suitable formalisms as well.

4 Control Integration

For an integration that exceeds the plain data format level, we define uniform interfaces of *adaptors* for tools. The notion of an adaptor illustrates that some functionality is plugged into the environment, no matter if the adaptor itself implements it, or if it just realizes a wrapper for another tool. We distinguish the following kinds of adaptors:

- *Model source adaptors* connect external modeling tools and techniques. They translate tool specific data into a known format and, by that, create initial formats from MSLs. They also keep track of whether the tool specific data is outdated, and formats derived by it need to be invalidated.
- *Function adaptors* implement forms of analysis or transformations and, by that, transform formats and/or derive new formats from existing ones.

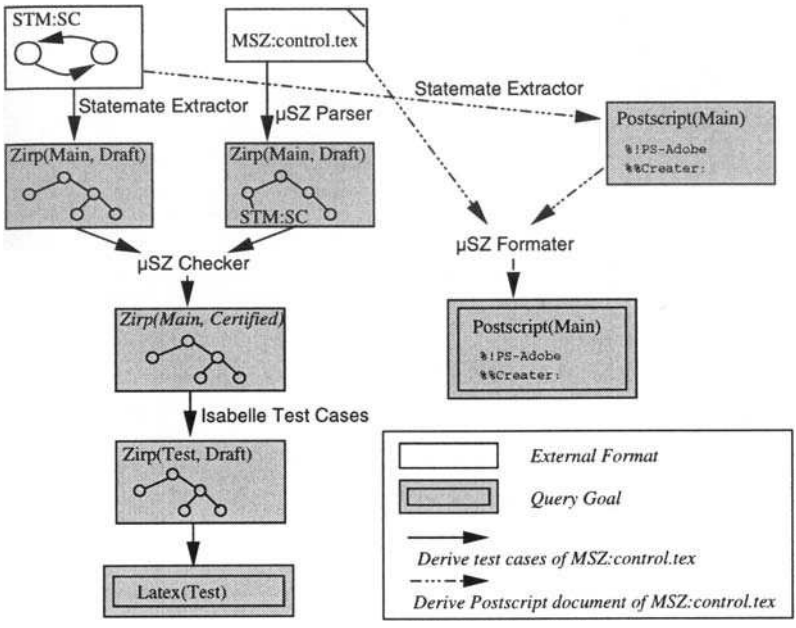


Fig. 3. Tool Chains: Driving of Format Queries

On registration, a model source adaptor tells the environment which tool-identifier (part of on MSL, such as `STM`) it serves, and which format types it can provide. A function adaptor is registered with the set of required input formats, and the output formats it will produce on success. This information enables a *data-driven* control integration. The basic scenario is that a user (or tool) requests a certain data format for an MSL from the environment; necessary actions to establish this format are then automatically performed by the environment.

Fig. 3 illustrates the process to resolve such requests. The formats ZIRP, Postscript and LaTeX are involved. The format qualifier $Main$ addresses the main specification document, the qualifier $Test$ the test-cases derived for this document. The state $Draft$ is associated with unchecked, the state $Certified$ with type-checked abstract syntax.

Given a compound document identified by the MSL `MSZ:control.tex` with a behavioral sub-model identified by `STM:SC`, two requests are illustrated: get $Latex(Test)$ of `MSZ:control.tex`, that is a LaTeX-representation of the derived test cases; and get $Postcript(Main)$ of `MSZ:control.tex`, that is a postscript version of the overall specification.

Two model source adaptors, the Statemate extractor and the μSZ parser for compound documents, are involved. The Statemate extractor translates Statecharts into the ZIRP format or into postscript. The μSZ type-checker is integrated by a function adaptor which merges the abstract syntax resulting from

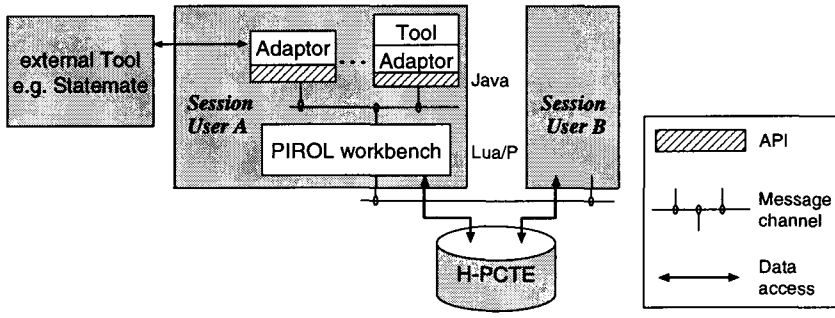


Fig. 4. Architecture of PIROL

the Statechart and the compound document, and certifies it. Test cases are calculated on the basis of a function adaptor which connects to the theorem prover Isabelle.

The example of the Isabelle test case adaptor illustrates some further aspects: First, this function adaptor will internally convert the ZIRP syntax into the HOL-Z encoding of Z in Isabelle [14], invoke an appropriate operation under HOL-Z, and convert the result back to ZIRP. Thus, hidden inside of a function adaptor, a mapping between uniform and proprietary data-formats may take place. Secondly, the generation of test-cases is of course not the only function provided by a powerful tool such as Isabelle. In fact, an adaptor does not necessarily coincide with a physical tool instance: there may be several adaptors connecting to the same tool instance. Each physical tool in turn needs to be started only once for each user session and runs as a background process waiting for requests from the environment. This takes into account that tools like Isabelle have a considerable start-up time.

Note that adaptors are generally considered to operate non-interactively. Analysis operations such as type-checking, which produce diagnostics, store the messages as a particular format of an MSL, which can be later on retrieved by browsing tools. Yet, some adaptors may require configuration parameters; the interactive definition of these parameters is generically realized by our environment.

By giving control over invocation of adaptors to the environment, editors and browsers are completely decoupled from any computations. Adaptors can be plugged in and out without modifying the visual tools.

5 Implementation

The ESPRESS tool environment is implemented by a multi-layer architecture. Fig. 4 depicts the main components: the H-PCTE persistent object storage, the PIROL workbench, and a Java-based application program interface (API) for formats and adaptors.

Persistent object storage. Basic integration of persistent data is managed by a central object management system (OMS). We employ H-PCTE [12] which implements a structurally object oriented database according to the ISO standard PCTE [18]. PCTE is especially designed as framework for software engineering environments providing services that exceed a plain database system.

PIROL Workbench. Control integration is handled by the PIROL workbench [13], which at the current state runs as a background-process for each user session. The workbench provides (possibly cached) access to the OMS and executes methods on persistent objects. This lifts persistent objects to full object orientation. OMS and workbench together form the full fledged active repository, that hosts all build algorithms for the driving of tool chains. These algorithms are implemented by means of the object oriented repository language Lua/P[2].

Formats, in particular abstract syntax trees, are treated as atomic repository objects which are opaque for the PIROL workbench. The desire to define an abstract syntax by the repository object model would yield one repository object from each node of the abstract syntax – which would be unbearable due to the overhead each repository object causes in terms of administrative data and access execution time. Even medium sized specifications easily produce syntax trees with several thousands of nodes. For this reason a second level of data modeling is deferred to the next architectural level, the Java-based tool API.

Tools technically converse only with one component: the PIROL workbench. This is the key to a scalable environment in which tools can be plugged in and out.

Java-based API. The definition of the format data types, the interfaces for tool adaptors, and general access to the repository is provided by an API given in Java (resp. Pizza for abstract data types). This API provides a complete abstraction and specialization for the concerns of ESPRESS of the underlying repository architecture. Repository objects are thereby interfaced by according Java proxy objects. The message-based coupling of Java with the PIROL workbench is done by means of Java's JNI (Java Native Interface).

The JNI also supports the implementation of adaptors for tools which provide a C interface, such as Statemate and its so-called dataport. Since Java provides a powerful general programming library including reflection, other forms of adaptors which base on textual exchange formats are easily implemented as well. For example, the Isabelle adaptor uses a generic parser/unparser for mapping ZIRP to SML Terms, and is implemented by reflection over the ZIRP format type. Reflection capabilities have also been used to construct a *binding compiler* which produces strongly-typed interfaces to Java classes for the functional language OPAL; this way, the adaptor for the μSZ type-checker ESZ, written in OPAL, has been realized. Moreover, an IDL mapping for Java and a Java ORB are available, such that Java-based adaptor implementations could connect to the

[2] Lua/P is an instance of the Lua [11] language family specially adapted for the PIROL repository

CORBA world. Last but not least, adaptors and other tools can be written from scratch in Java resp. Pizza itself – this is supported by Java and Pizza libraries tailored for manipulating formats.

Existing Adaptors. Some of the tools currently integrated by according adaptors are briefly sketched below:

- *XEmacs (→ LaTeX).* An elaborated XEmacs mode for visual editing and browsing of compound specification documents, which also realizes a frontend for most of the currently integrated functionality. In the tool chains, it is integrated as a model source adaptor, which imports compound documents in LaTeX format from the filesystem into the environment.
- *Statemate(→ ZIRP/Postscript).* The Statemate tool is integrated by a model source adaptor that resolves Statemate MSLs and extracts Statemate models. It reads its data from the Statemate API ("dataport") resp. the Statemate databank format and provides Postscript or ZIRP output.
- *ESZ (LaTeX→ ZIRP).* A function adaptor which parses and typechecks μSZ specifications.
- *Isabelle/HOL-Z (ZIRP → ZIRP).* A function adaptor that connects the generic deduction system Isabelle to the environment. More specifically we employ HOL-Z which instantiates Isabelle for Z/μSZ. This adaptor implements several analysis, such as DNF-calculation for test-case generation, precondition calculation of Z schemas, guard analysis for combined Statechart/Z specifications, and more.
- *Pretty Printer (ZIRP → LaTeX).* Probably the simplest function adaptor. It converts the abstract syntax format ZIRP into a format holding LaTeX. This is in particular useful for re-integrating the result of a automatic transformations – such as test-case generation – back into a specification document.
- *ZAP (ZIRP → Java).* A function adaptor that compiles a functional subset of Z into Java code.

Meta modeling issues. Referencing persistent objects is one of the basic facilities of a repository. MSLs add an new referencing mechanism. An MSL comprises a tool identification and a local address that is interpreted by the corresponding tool. Once it is resolved, an MSL refers to a persistent unit in the repository, which is subject to common caching and outdating mechanisms. Requests to resolve an MSL also specify a required format. Reference resolution and provision of this format are technically separated in order to grant maximum flexibility in reusing existing information and converting to all desired formats.

Granularity of data modeling is a crucial issue for tight data integration. The approach of ISO PCTE to merge concepts for fine grained objects into PCTE [19] must be considered unsuccessful because of the large difficulty to implement the standard. Our approach draws a clear line between the medium grained repository object model (given in Lua/P) and the very fine grained abstract syntax (given in Java/Pizza). The first level grants the workbench access to all information needed by environment services. The second level provides the basis

for close cooperation between tools, yet pertaining a sufficient performance of the system.

6 Related Work

Few projects approach the task of integrating tools for formal methods and existing commercial CASE tools in all aspects discussed in this paper. The approach of the UniForM Workbench [15] is quite similar to our's. Whereas our integration languages Java and Lua/P are object oriented, UniForM employs a functional language (Haskell) for this task. Also the architecture differs: UniForM shows a greater number of general components, called managers. We implement the functionality of these managers by just one component, the workbench, thus reducing the number of interfaces. A closer comparison of these two system should trigger further interesting evolution.

Other approaches like the Concurrency Factory [4] or AutoFocus [10] present more homogenous environments, with most tools written from scratch and less emphasis on an active repository as a component for integration of persistent data and control.

7 Conclusion

We have presented the design principles of a tool environment which can integrate heterogeneous formal and non-formal modeling as well as analysis and transformation techniques. Our approach bases on the discernment that we have to reuse and integrate existing tools to bring formal methods to practice. We emphasize that the objective of a tool environment for formal methods has to take advantage of existing, sophisticated tools rather then reinventing them – which would be intractable in todays praxis.

Our approach anticipates that we have to live with "rough edges" if we want to integrate tools for the application of formal methods in an environment which has a chance for acceptance in industrial praxis. A major principle of our work is to grind these edges, instead of trying to eliminate them. In particular, we give solutions on how external modeling techniques can be integrated in our framework – let them be just non-integrated text-editors such as the "vi" or comprehensive CASE tools such as Statemate.

Despite of all constraints given by external tools a high level of integration has been reached. This contributes to the convenience of tool support for formal methods. By using an open repository–based environment as integration platform, separation of concerns is enforced and interdependencies between tools are reduced. Tools request services only from one component (the workbench) but indirectly make use of the functionality of the whole environment. This separation of concerns helps to add new features to the environment without disturbing existing functionality. *Version- and access control* are tasks of the repository proper. Universal tools for handling versions and configuring access

control easily cooperate with all other tools in the environment. The same holds for general repository browsers, printing utilities etc.

We have shown how the integration of data, control and presentation is implemented. In addition to these aspects the ECMA reference model [5] mentions "process integration" and "framework integration". Both blend well with the concepts introduced so far. The PIROL repository object model is designed also for support of the *development process* in terms of organization and methodology. The term *framework integration* captures services for common administrative tasks like tool configuration and administration of users and projects. Tool configuration is handled by the tool libraries in conjunction with the workbench. All other administration is orthogonal to any other service and support can be added easily.

The environment represented in this paper has been implemented and applied in the course of the ESPRESS project. Currently, we have integrated the foreign tools Statemate, XEmacs, Isabelle, FDR, and the ESPRESS specific tools ESZ (a Z/μSZ type checker) and ZAP (a compiler for Z). The applicability of our approach has been demonstrated in two case-studies with industrial background. One case-study is the specification of a traffic light system [1], the second [7] is the specification of an intelligent cruise control.

References

1. R. Büssow, R. Geisler, and M. Klar. Specifying safety-critical embedded systems with statecharts and Z: A case study. In Egidio Astesiano, editor, *Proc. of the 1st Intl. Conf. on Fundemantal Approaches to Software Engineering – FASE'98*, volume 1382 of *LNCS*, pages 71–87. Springer-Verlang, 1998.
2. Robert Büssow, Robert Geisler, Wolfgang Grieskamp, and Marcus Klar. The μSZ notation version 1.0. Technical Report 97–26, Technische Universität Berlin, Fachbereich Informatik, December 1997.
3. Robert Büssow and Wolfgang Grieskamp. Combining Z and temporal interval logics for the formalization of properties and behaviors of embedded systems. In R. K. Shyamasundar and K. Ueda, editors, *Advances in Computing Science – Asian '97*, volume 1345 of *LNCS*, pages 46–56. Springer-Verlag, 1997.
4. R. Cleaveland, J.N. Gada, P.M. Lewis, S.A. Smolka, O. Sokolsky, and S. Zhang. The concurrency factory — practical tools for specification, simulation, verification, and implementation of concurrent systems. In *Proceedings of the DIMACS Workshop on Specification of Parallel Algorithms*, Princeton, NJ, May 1994.
5. Reference Model for Frameworks of Software Engineering Environments – ECMA TR/55 3rd edition. Technical report, European Computer Manufacturers Association (ECMA), June 1993.
6. Formal Systems (Europe) Ltd. *Failures Divergence Refinement, FDR2 User Manual*, 1997.
7. Wolfgang Grieskamp, Maritta Heisel, and Heiko Dörr. Specifying embedded systems with statecharts and Z: An agenda for cyclic software components. In Egidio Astesiano, editor, *Proc. of the 1st Intl. Conf. on Fundemantal Approaches to Software Engineering – FASE'98*, volume 1382 of *LNCS*, pages 88–106. Springer-Verlang, 1998.

8. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
9. David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16 No. 4, April 1990.
10. F. Huber, B. Schätz, A. Schmidt, and K. Spies. AutoFocus - a tool for distributed systems specification. In *FTRTFT'96 – Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135 of *LNCS*, pages pp. 467–470. Springer Verlag, 1996.
11. R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.
12. Udo Kelter. H–PCTE — a high–performance object management system for system development environments. In *Proc. COMPSAC '92*, pages 45–50, Chicago, Illinois, September 1992.
13. Boris Groth; Stephan Herrmann; Stefan Jähnichen; Wilfried Koch. Project Integrating Reference Object Library (PIROL): An object–oriented mutliple–view SEE. In *Proceedings of the 7th Conference on Software Engineering Environments (SEE'95)*, Noordwijkerhout, Holland, April 1995.
14. Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher-Order Logics*, LNCS 1125, pages 283–298. Springer-Verlag, 1996.
15. B. Krieg-Brückner, J. Peleska, E.-R. Olderog, D. Balzer, and A. Baer. Universelle Entwicklungsumgebung für Formale Methoden (UniForM Workbench). Informatik Bericht 8/95, Universität Bremen, 1995.
16. Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.
17. L. C. Paulson. *Isabelle – A Generic Theorem Prover*. LNCS 828. Springer-Verlag, 1994.
18. ISO/IEC 13719-1: Portable Common Tool Environment (PCTE). Abstract specification, International Organization for Standardization (ISO), 1995.
19. Amendment 1 to ISO/IEC 13719-1: Fine-grain object extensions. Technical report, International Organization for Standardization (ISO), 1995.
20. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

Integrating Domain Specific Language Design in the Software Life Cycle

Philipp W. Kutter[1], Daniel Schweizer[2], and Lothar Thiele[1]

[1] Federal Institute of Technology, CH-8092 Zürich
{kutter, thiele}@tik.ee.ethz.ch
[2] United Bank of Switzerland, CH-8021 Zürich
daniel-sja.schweizer@ubs.com

Abstract. Domain Specific Languages help to split the software live cycle in different independent cycles. While the use of the newly created language is just an additional tool in the established cycle, the language live cycle is independent and opens the doors for the application of formal methods. We report on an industrial case study, where a driver specification language has been designed, formally specified, and finally an implementation has been generated from the specification. Using Abstract State Machines and Montages for the language specification, it was possible that the industrial partners learned how to maintain and extend the language specification. On the other hand the formal semantics of the method allows to apply different verification-oriented methods to the artifacts.

1 Introduction

Today's software and hardware systems are characterized by a composition of complex subsystems and complex interactions between them. These subsystems are very often based on different models of computation and data. Interfaces among these heterogeneous subsystems and other problems of inter-operability are major sources of complexity.

For example, in a data-warehouse scenario there is the need of connecting different information sources, such as different data bases and different data sources (e.g. Internet, files, data bases, user input/output). A similar situation can be found in the specification and implementation of heterogeneous electronic systems. No single model of computation is appropriate for all application areas. The correctness of interfaces between components modeled with different paradigms are of major importance.

Traditionally there have been two major approaches to solve these problems:

Problem–specific: The interfaces (wrappers) between the heterogeneous components are implemented by hand in a tedious process. In addition the interfaces are written without reference to the underlying model of computation which is error prone.

General purpose: Frameworks have been provided for the specification of all system components and their interaction. This approach has proven to be not feasible in several application domains as these general purpose frameworks are too complex to manage and to use by domain experts.

Consequently there is the need for new methods and tools to design heterogeneous components and their interaction.

Pretty early came up the idea, that little languages, tailored towards the specific needs of a particular domain, can significantly ease building software systems for that domain [Ben86]. This domain-specific approach, situated between the traditional problem-specific and general purpose solutions, enables an interaction between the different domains on the language level, and not on the implementation level. Already [HB88] notes that if a domain is rich enough and program tasks within the domain are common enough, a language supporting the primitive concepts of the domain is called for, and such a language may allow a description of few lines long to replace many thousand lines of code in other languages. This idea is elaborated in the FAST process, a software–engineering methodology advocating the use of domain–specific languages [Wei97]. Central to FAST is the process to identify a suitable domain of problems, and to find abstractions common to the domain. These abstractions are used to produce family members in an orthogonal way. As a next step, a language is designed for specifying family members. The syntax of the language is based on the terminology already used by the domain experts, and the semantics is developed in tight collaboration with them. The goals are to bring the domain experts in the production loop, to respond rapidly to changes in the requirements, to separate the concerns of requirement determination from design and coding, and finally to rapidly generated deliverable code and documentation. Central for the industry is the possibility to capture and leverage the expertise of the involved domain–engineers.

For the definition and implementation of DSLs, powerful tool support is needed. Both the compiler construction community and the language specification community are now advocating to use their tools for DSLs. Advantages of compiler construction tools (e.g. Cocktail [GE90], Eli [GHL+92], or PCCTS [PDC92]) are efficient implementations and scalability to large languages, while advantages of language specification tools (e.g. Centaur [BCD+87], ASF+SDF [Kli93], or Gem-Mex [AKP97a]) are very abstract specifications, that are easy to maintain and reuse. In an industrial setting, it has been shown that the advantages of both approaches can be combined by using operational language specifications as basis for DSL definition and implementation. In the following the work directly related to this approach is summarized.

In particular, the results present in the paper can be summarized as follows:

- A complex data warehouse problem at Union Bank of Switzerland is analyzed and the corresponding requirements for a solution are derived.
- Strictly based on these requirements, different possible solutions are compared and the use of domain-specific languages is selected as the appropriate approach.

- Using formal methods and corresponding software tools, difficulties in the application of DSLs have been resolved. In particular, the approach is based on abstract state machines, see [Gur95], and on a domain-specific structuring mechanism Montages, see [KP97b].
- The above formal methods have been integrated into the software life cycle.

2 Application

Union Bank of Switzerland (UBS) is the world second largest bank. Numerous large scale databases are used to provide information to cope with the challenges of daily business as well as for long term strategic decisions. Data relevant to both fields reside on many different data bases from different manufacturers distributed all over the world. Most of them are relational data bases up to size of 1 Tbyte. They are grown historically and thus expose very heterogeneous structures and data models, respectively. There is nothing like a company wide data model and after the merger with Swiss Bank Corporation (SBC), more then ever, hardly anyone believes that there will be one in the future. One centrally located database is technically and politically not feasible and also from the viewpoint of accessibility and availability not desirable. Until recently, data access was very cumbersome, especially in cases where information from different sources has to be combined in a single report. Due to the lack of suitable tools on the market, the CUBIX [Mar97] project was started two years ago. Since September 1997 the system is up and running, serving a rapidly growing community of several dozens users, mainly in controlling departments. CUBIX now provides access to information including profit centers, products, customers and accounts. In the following we sketch the CUBIX architecture, and summarize the requirements for rapid connection of new data-bases, that led to our application.

2.1 CUBIX

CUBIX provides an infrastructure to build and maintain so called "virtual" data warehouses, needed to do "Online Analytical Processing" "Virtual" means, that it is not necessary to copy the relevant information to a single huge database. OLAP has evolved as users' needs for data analysis have grown. It provides executives, analysts and managers with valuable information via a "slice, dice and rotate" method of end user data access, augmenting or replacing the more complicated relational query. This slice and dice method gives the user consistently fast access to a wide variety of views of data organized by key selection criteria that match the real dimensions of the modern enterprise. Although CUBIX provides a multidimensional view [CCS93] (see the data cube with three dimensions on top of Fig. 1), the data still reside on existing relational databases as they did before. A CUBIX data warehouse may consist of several data-marts, each containing information about distinct aspects of the company. Data-marts are represented as multi-dimensional cubes. Several cubes may share some dimensions. The user doesn't have to care about the location or kind of data bases he

uses. Reports spanning over multiple existing data bases are as easy to get as such from a single one.

2.2 Cube Drivers

One important component of the CUBIX system are drivers connecting the new multi-dimensional data model to the existing relational data bases. The core task of such a driver is to correlate the dimensions in the cubes to the tables and attributes in a relational data base. This is illustrated in Fig.1 showing how a tuple of coordinates is translated to a SQL-query retrieving the corresponding data in the relational data base.

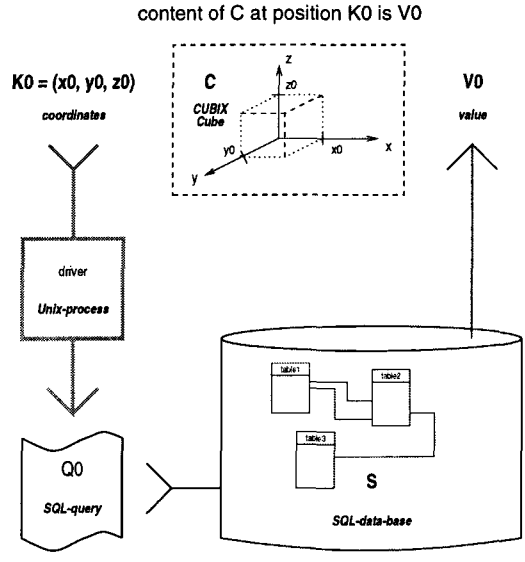


Fig. 1. Cubes and tables connected by a driver.

2.3 Physical Representation in Relational Databases

The basic components of the multidimensional model (MDM) are facts, dimensions and attributes. The first two components, facts and dimensions, are represented physically in a relational database as tables. In the simplest MDM, the basic star schema, they are the only tables. In the star schema, each dimension is described by its own table, and the facts are arranged in a single large table, indexed by a multi-part key that is made up of the individual keys of each dimension. There are many variations on the simple star schema, but they all share the basic concept of fact tables and dimension tables. In many cases, not

all facts share the same dimensionality, and multiple fact tables may be used. An example is selling price, which may not vary between markets or customers, or the Federal Income Tax rate. When dimensions are large (high cardinality), it often makes sense to split the dimension tables at the level of the attributes, also known as the "snowflake schema". Storing aggregations and derivations yields even more exotic combinations, sometimes referred to as "decomposed stars" or "constellation schema".

2.4 Requirements for Connecting New Databases

As mentioned earlier, our data bases grew historically, hardly any of them conforms to the star or snow–flake scheme. Thus the connection of a new data base requires a distinct cube driver to be implemented (or at least configured). So far, the implementation of a new cube driver took several days and had to be done by a highly trained computer scientist. Driven by the fast-growing needs for information within UBS and further accelerated by the recent merger with SBC, the number of data bases to be connected grows rapidly. Thus the rapid connection of new databases is crucial for the success of CUBIX and the requirements for connecting data-bases can be summarized as follows:

(1) The Connection of existing data bases must be possible in less than 1 week. Only a significant gain in time to market justifies the use of a new technology.
(2) Data base administration staff should be able to connect their data bases to the CUBIX data warehouse. Training to do this should be less than two days.
(3) Although CUBIX is already in use in daily business, development continues. Thus, if some changes of the CUBIX implementation are required, all drivers should be automatically adaptable.
(4) To benefit from new versions of underlying data base management Systems, an adaption of the respective cube drivers may be required (e.g. to employ new query optimizations mechanisms).
(5) The Cubix developers must be able to understand all aspects of the solution chosen for connecting data bases. To be able to incorporate future optimizations, they need full control over the used technique. The solution must be easy to document, easy to extend to new problems, and easy to learn to new people in the team.

3 Methodology

Three possible approaches to meet the requirements as described in section 2.4 have been investigated.

- Currently, the interfaces between the heterogeneous data sources are implemented by hand. Besides the fact that this problem-specific process is error prone, it violates the requirements (1), (2), (3) and (4).

- A typical academic approach is to build a general-purpose framework for the specification of all system components and their interaction. With such a system, both multi-dimensional and relational data-models can be represented including their interaction. Consequently, the requirements (1), (3) and (4) can be satisfied. On the other hand, these frameworks are too complex to manage and to use by domain experts, see (2) and (5).
- In a domain-specific approach, a small language specifies the interfaces between the heterogeneous data sources. Obviously, handling (3) and (4) is particularly simple as only the semantics of the interface specification language must be adopted if CUBIX or a data-base management system changes. The specification of the DSL should be easy to understand, extend, and maintain such that domain experts are able not only to use them, but also to change them, see (1), (2) and (5). To this end, formal methods and tool support has been developed and integrated in the software life cycle at Union Bank of Switzerland.

In the given application, we decided to develop a driver-specification language, called Correlation Modeling Language (CML). Each program in CML describes a driver. Having decided to follow this approach, additional requirements came up, mainly connected with the old requirement (5).

(5.1) The syntax and semantics of the language must be unambiguously defined.
(5.2) The language definition must be easy to read and understand.
(5.3) The language should be easy to extend.
(5.4) Changes in the definition should be implementable without specific knowledge, e.g. in compiler–construction techniques or formal methods.

Requirements (5.4) and (5.2) rule out traditional implementation techniques. As alternative remain sophisticated compiler construction tools and programming language semantics formalisms. Although at a first glance, compiler construction tools seem to be much more appropriate for an industrial context, they are not suited for our purpose. They allow to develop very efficient compilers for new languages. The involved techniques require a deep understanding of compiler construction, which contradicts requirements (5.2), and (5.3).

Using a programming languages formalism that allows to generate an implementation from the specification, the driver generation process can be split in three phases as depicted in Fig. 2. In the first phase, a specification of CML (written for instance in the structuring formalism Montages, see section 3.1) is fed to a rapid prototyping software tool (here for instance Gem-Mex, see section 4). The rapid prototyping tool generates an interpreter of CML. Using this interpreter a driver specification can be executed. Figure 1 is a more detailed presentation of phase three in Fig. 2.

3.1 Formal Semantics for Domain Specific Languages

A number of semantics formalisms have been proposed, and they have been successfully used to investigate single constructs of languages and to discover flaws

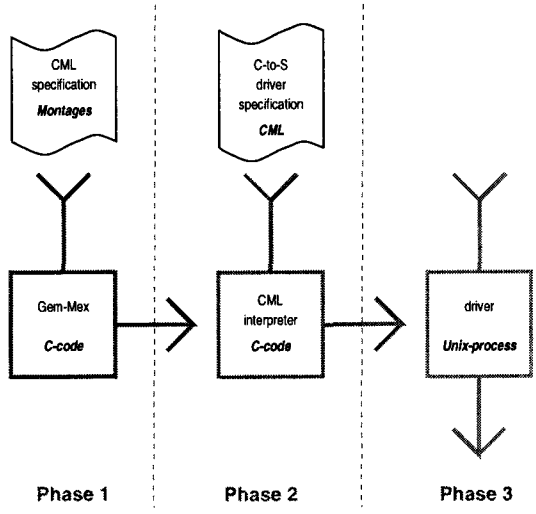


Fig. 2. The DSL process

in the design of languages. They had a considerable influence on the design of functional and logic programming languages. Recently most of them have been applied to DSLs as well. The approaches can be split in operational [Hen91], denotational [Sto77], and axiomatic [Dij76,Hoa69] ones. In operational semantics, the meaning of a well-formed program is the trace of computation steps that results from processing the program's input. In denotational semantics the is a mathematical function from input data to output data. The states of a computation must be encoded in the output data. Finally in axiomatic semantics the meaning is a logical proposition, that states some property about the input and outputs, respectively the state before or after a computation step.

Unfortunately the mission to provide a semantical analogue to EBNF was never accomplished. Although Scott-Strachey-style denotational semantics [SS71] introduced 25 years ago the revolutionary idea to extend BNF to semantics, it has never been accepted by programmers, since denotational semantics of a realistic language is often too dense to read and too low level to modify [Sch97]. In general the disadvantage of the proposed approaches is that they involve a relatively heavy mathematical machinery. As discussed above, for the present application of DSLs there was a strict requirement (5) for a formal semantics, involving a minimum of mathematical machinery.

Montages [KP97b,AKP97c] use Gurevitchs Abstract State Machines(ASMs) [Gur95] to extend BNF to semantics. Syntax, static analysis and semantics, and dynamic semantics are given in an unambiguous and coherent way by means of semi-visual descriptions. The specifications are similar in structure, length, and complexity to those found in common language manuals. Montages are designed to provide a theoretical basis for a number of activities from initial language

design to prototyping. In order to meet requirement (5) Montages are based only on concepts well know to industrial programmers:

- EBNF for syntax definition
- control/data-flow graphs for static analysis
- simple predicate logic for static semantics
- imperative pseudo code for dynamic semantics

Formally, the imperative pseudo code are ASM rules, and the static aspects of Montages are formalized with ASMs too. Thus intuition of Montages is based on the above listened concepts, while formal semantics is given with ASMs. Such a formalization is in turn understandable as imperative pseudo code, and can thus be used as documentation for the industrial programmers. In the following we introduce ASMs and Montages.

Abstract State Machines ASMs [Gur95,Hug] (formally called Evolving Algebras) are an operational specification formalism. In short, ASMs are a state–based formalism in which a state is updated in discrete time steps. Unlike most state based systems, the state is given by an algebra, that is, a collection of functions and a set of objects. The state transitions are given by rules that are fired repeatedly.

Objects: in each ASM the set of existing objects $\mathcal{O}$ [1] plays a fundamental rôle. Each object has its identity, independent of the value of its attributes. For example we may express that all objects satisfy predicate P

$$\forall o : \mathcal{O} : P(o)$$

Functions: Constants, Operations, Sets, Variables, Attributes, Relations, Records, Arrays, Data Structures correspond all to n-ary functions. As higher programming languages abstract from registers, ASMs abstract from the differences of the listed concepts.

Constants and *Operations* are static functions without respectively with arguments. Static means that they do not change their definition over time. For convenience the constants *true* and *false* are predefined denoting two different elements of $\mathcal{O}$ and the logical operations are predefined. In addition a third object is denoted by the predefined constant *undef*.

Sets [2] or *Types* in the sense of collections of objects are modeled by functions from O to $\{true, false\}$. Such a function delivers *true* for all members of a set (instances of a type), and false otherwise. The set or type consisting of *true* and *false* is called *Boolean*. A type T corresponds thus to a function

$$T : \mathcal{O} \rightarrow Boolean$$

[1] in traditional ASM terminology objects are called elements and $\mathcal{O}$ is called the super-universe

[2] traditionally ASM literature speaks about universes

Types are used to describe domain and codomain of functions. But there is no fixed type system as for instance in many-sorted or order-sorted algebras. In [Kut98] simple macros allowing to use set operations are defined.

Variables are 0-are functions, that can be redefined. *Attributes* are unary functions, that can be point-wise redefined. *Relations* are functions with more than one argument and Boolean as codomain. *Records* are unary functions from field–names to field–values, *arrays* are unary functions with a range of positive integers as domain. In general *data structures* are n-ary functions that can be redefined point wise.

The n-ary function is thus a unique concept, used to represent all sorts of functional dependencies, relations, storage, and data structures.

Dynamics: Given a state of an ASM consisting of $\mathcal{O}$ and definitions of the declared functions[3] the system evolves by repeated execution of calculation steps. Calculation steps are defined with *transition rules*. The fundamental construct of transition rules is the *update* or redefinition of a n-ary function at one point.

$$f(t_1, \ldots, t_n) := t_0$$

redefines f at point $v_1, \ldots, v_n$ to v_0, where $v_0, v_1, \ldots, v_n$ are the values of $t_0, t_1, \ldots, t_n$ in the current state. The update may be guarded, and the simple guard construct can be generalized to if-then-elsif-else and case constructs.

Abstracting Calculation Steps Unlike sequential, imperative languages ASMs allow to specify in a declarative way, *what* changes from one state to another rather than *how* this is done sequentially. This is achieved by means of *parallel composition* of updates. Swapping the values of variables x and y is done by executing the rule

x := y y := x

which naturally corresponds to the equation system $\{x' = y, y' = x\}$ relating as usual next-state (primed versions x' and y') with the current state. Arbitrary sequentialisation can be achieved by means of abstract program counters.

ASMs where used to successfully describe the dynamic semantics of programming languages like Prolog [BR94], C [GH93], C++ [Wal95], Occam [BDR94b], Oberon [Kut97], Java [BS98], and SDF [GK97].

These specifications have been used to verify the correctness of compilation, e.g from Prolog to WAM [BDR94a], Occam to Transputer [BD96], and C to DEC–Alpha processors [ZG97]. An up–to–date overview on case–studies with ASMs can be found on [Hug].

An important differences of the dynamic semantics specifications with ASMs to those using classical approaches, is the way how the parse tree is formalized. The classical way is to consider the parse tree as an element of the term–algebra generated by the context free grammar. In newer approaches like

[3] $\mathcal{O}$ (the super-universe) together with interpretations of the function symbols builds an algebra, thus the states of ASMs are algebras.

[Gan85,Gur88,Ode89,PH91] and the mentioned ASM dynamic-semantics descriptions, each parse tree is formalized as an algebra, whose carrier sets contain the occurrences of the nodes. Based on this approach, control/data-flow graphs, can be directly specified and need not to be encoded with tables or continuations. In [PH97] functional specifications of static analysis and semantics are combined with ASMs for dynamic semantics. The case-studies show, that the method scales to languages like C, and that pretty efficient interpreters can be generated.

Montages , as described above can be seen as an application specific structuring mechanism for ASMs. A language description is given by a collection of Montages, which is hierarchically structured according to the rules of the corresponding context-free grammar given in EBNF. Each Montage is a "BNF-extension-to-semantics", that is a self contained description in which the semantic properties of a given construct are formally defined.

In Fig. 3 the Montage specification of the select snow–flake construct of the developed driver specification language CML is given. Syntactically the construct is built by the keyword "SNOW-FLAKE" and a list of value-type (given by a number $Number), attribute-name ($Ident) pairs. pairs. The purpose of the construct is to choose the corresponding attribute-name, for the currently requested value type (*RequestedValueType*) and to assign it to the attribute Code of the SelectSnowFlake-node. For completeness, the Montage of the TypeAttrPair is given as well.

The topmost part of the Montage is the production rule defining the context–free syntax. Below is a graphical representation of a pattern in the parse tree, and of the control and data flow graph. Inner nodes of the parse tree are represented with boxes and leaves with ovals. Nested boxes are used to represent nodes on lower levels of the parse tree. The solid and dotted arrows denote the data and control flow, respectively. Control flow arrows are labeled by means of boolean predicates which determine through which edges the control flows from one state to the next, e.g. in our example the predicate *RequestedValueType = Self.S-Number* indicates that the control is passed from a TypeAttrPair-node to a SelectSnowFlake-node whenever the number-value is equal to the requested value type. As in object-oriented languages, the Self is default, and can be omitted. For the sake of simplicity, we allow as well to omit the label in certain situations:

- If the unique outgoing control arrow of a node has the label *true*, this label may be omitted.
- If two control arrows c_1 and c_2 leave a node, and c_1 is labeled with the negation of c_2's label, one of the labels may be omitted.

The control flow arrows I (initial) and T (terminal) are special arrows which serve to plug together the local flow-information to the global one. For illustration, the global flow graph for the following example program

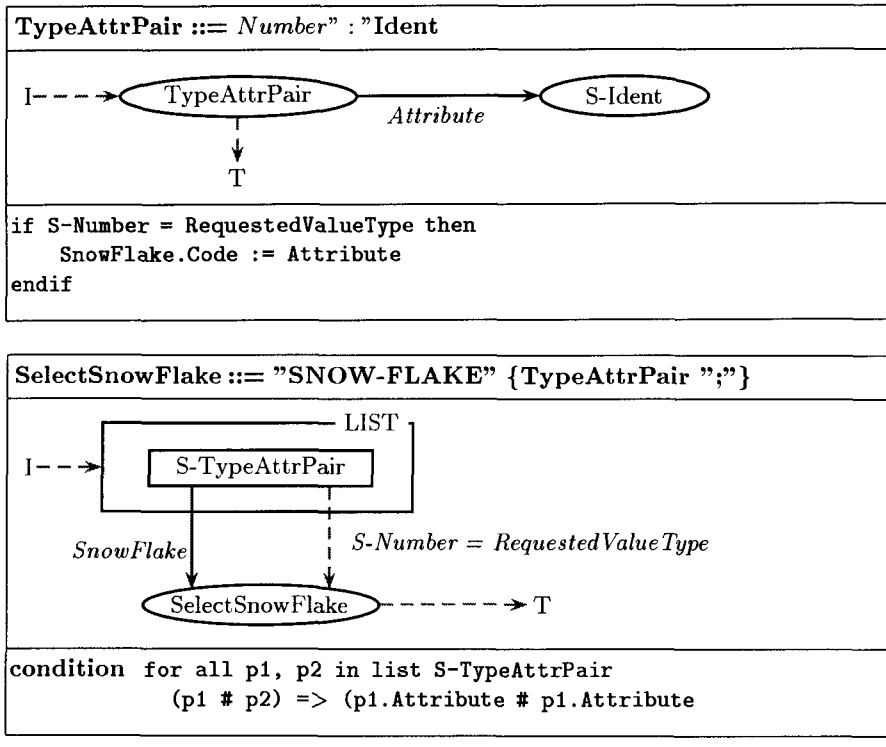


Fig. 3. The select snow-flake construct

```
SNOW-FLAKE
  1: Revenue;
  2: Amount;
  3: Risk;
```

is given in Fig. 4. To simplify the picture, the SnowFlake arrows are not drawn.

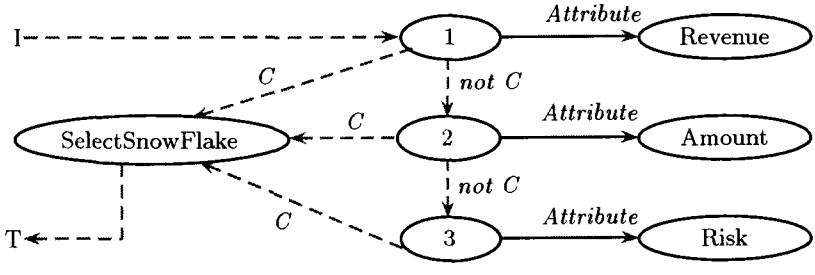


Fig. 4. A global flow graph, where C $\triangleq$ (number = RequestedValueType)

The plugging mechanism is defined as follows.

- The functions Initial and Terminal are initialized as identity. The I and T arrows are used to redefine them as follows:

  Self.Initial := Self.S-I.Initial
  Self.Terminal := Self.S-T.Terminal

  where S-I is the selector to the descendant pointed by the I arrow, and S-T is the selector to the descendant being source of the T arrow.
- A solid data–flow edge links the source with the terminal task of the target. This is useful, since the result of a construct is typically available at the end of its evaluation, e.g. at its terminal task.
- A dotted control flow edge links the terminal task of its source with the initial task of its target.

The third part of a Montage contains the static semantics (condition), for instance in the SelectSnowFlake (Fig. 3, the TypeAttrPair-nodes must have different number-values. The designer may make use of full first-order logic to express context sensitive constraints. The last part contains the dynamic semantics rules, in the example the SelectSnowFlake has no dynamic semantics rule, and the dynamic semantics rule of the TypeAttrPair sets the Code attribute of the SnowFlake to the corresponding attribute.

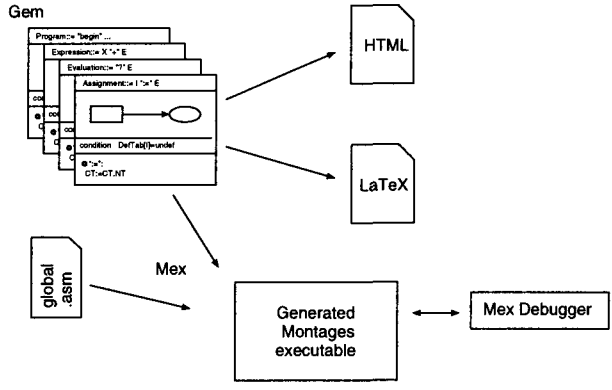


Fig. 5. The General Structure of the Gem-Mex System

4 Tool Support

The development environment for Montages is given by the Gem-Mex tool [AKP97b], whose architecture is depicted in figure 5. It is a complex system which assists the designer in a number of activities related with the language life cycle, from early design to routine programmer usage.

It consists of a number of interconnected components

- the Graphical Editor for Montages (Gem) is a sophisticated graphical editor in which Montages can be entered; furthermore high-quality documentation can be generated;
- the Montages executable generator (Mex) which automatically generates correct and efficient implementations of the language;
- the generic animation and debugger tool visualizes the static and dynamical behavior of the specified language at a symbolic level; source programs written in the specified language can be animated and inspected in a visual environment.

The whole development of a programming language can be supported with an effective impact on the productivity and robustness of the design. The designer can enter the specification, browse it and especially maintain it. Specifications may evolve in time even in a non-monotonic way since modifications can be localized within very neat boundaries. By doing so, different experimentation can take place with different versions of the syntax and semantics of the specified language in a very short time.

Besides the pure editing functionality, Gem can be used to generate documents suitable for specification presentation. Experience suggests how lack in documentation is a dangerous bottleneck for the consistency and coherence of a project. Both, paper and online presentation of the language specification are automatically generated by Gem:

- LaTeX documents illustrate the Montages and the grammar; such documents are easily customizable for the non-specialist user;
- HTML versions of the language specification allows to browse the specification and retrieve pieces of specification.

Moreover, intelligibility is enhanced by means of "literate specification" techniques directly supported by Gem. Formal parts of the specification can be substituted with textual elements by means of a "literate programming" tool integrated in the system. "Literate specification" means that the Montages text fields may contain references to other parts of the formalization specified outside of the Montages modules.

For the collaboration with industry, it was important, that Gem-Mex is integrated with the programming language C, e.g. the generated interpreter can be called directly from a C-program, and C-programs can be called inside Montages descriptions. In Fig. 2 we see how Gem-Mex is integrated in the driver development process. For the application to CML, the by Gem-Mex generated interpreter has shown to been efficient enough and is now directly running in the operational version of CUBIX at UBS. To provide more efficient implementations we continue to work on the integration of Montages with the Verifix approach for correct compiler construction [ZG97].

5 Integration in Software Life Cycle

Before the introduction of CML the development of driver has been tangled with both the current state of CUBIX and the evolution of the underlying DBMS. To apply formal methods directly in such a frame work would not be possible. Through the introduction of a DSL, the life cycles of the driver software and the other components are less intertwined and the introduction of formal methods for the development of CML was possible even guaranteeing that the final user needs only handle a very simple language. Such a user can concentrate on the correlation of his data-base and a cube, without worrying about CUBIX or the underlying DBMS. The CUBIX team, maintaining the CML specification, can easily adapt the specification to changes in their system or to changes in the DBMS. Finally improvements of Gem-Mex are directly beneficial for the already specified drivers, and for the maintenance process of CML. As well, if the drivers are ported to another system, only Gem-Mex has to be adapted.

6 Conclusions

The application of the afore mentioned methods and tools in the CUBIX project led to more universal and more flexible solutions for connecting relational data bases to the CUBIX data warehouse. The domain specific language CML together with the tools supporting its application reduced the effort and skills necessary to rapidly establish and maintain data base connections significantly.

The advances of using Gem-Mex to design, specify, implement, and document CML are:

- **Simplicity** Only simple imperative updates and drawings of control and data flow graphs are used to specify the language. Only 28 different symbols (for variables, macros, e.t.c.) are used in addition to the symbols introduced by the EBNF syntax-rules. The whole textual specification of static and dynamic semantics is 92 lines long.
- **Modularity and Extendibility** The specification is split in 22 Montages. Each Montage specifies one language construct. Most information is local, and changes of one Montage do not affect the other Montages.
- **Self documenting** The simplicity, shortness, and modularity of the specification make it usable as documentation as well. Using again the Montages of CML, Gem-Mex generates HTML and paper versions of the language description. The Montages approach could be learned within one day by an industrial programmer, and allows the CUBIX developer to extend the language if new problems appear.
  item **Efficiency** From the Montages specifications of CML, Gem-Mex generates an interpreter. This interpreter is efficient enough to be used in the operational version of CUBIX. Calculating the SQL queries for a report with 100 result values, using a complicate driver specification takes about 1.5 seconds on an Ultra30, for a simple specification about 0.6 seconds.

- **Portability** Gem-Mex generates C-code, that can be compiled on arbitrary architectures and operating systems. Gem-Mex itself is only based on C and TclTk, which exist on all commercial platforms, like Windows, Windows NT, Unix. The tools can thus easily be integrated in a developer version of CUBIX.

It has been shown that abstract state machines [Gur95] structed with Montages [KP97b] are well suited to define domain specific languages. The formal specification of the language CML has been a starting point to successfully introduce formal methods into an industrial setting. In particular, formal methods specialists can read Montage models since the semantics of abstract state machines is a mathematical object. Domain experts need not learn other formal methods in order to validate the developed formalizations. Therefore, formal methods far above the complexity understandable by domain experts can be applied, if a Montage/ASM specification exists.

Acknowledgments We would like to thank to Matthias Anlauff, Alfonso Pierantonio, and Christoph Denzler for their help with Gem-Mex and their collaboration in the Montages project, and to Werner Brunner, Reto Marti, Stefan Pascheda and Matthias Seitz at UBS for their support and confidence.

References

[AKP97a] M. Anlauff, P. Kutter, and A. Pierantonio. Formal aspects of and development environments for montages. In *ASF+SDF'97*, Workshops in Computing. Springer, 1997.

[AKP97b] M. Anlauff, P.W. Kutter, and A. Pierantonio. GemMex-Homepage, 1997. http://www.first.gmd.de/~ma/gemmex/.

[AKP97c] M. Anlauff, P.W. Kutter, and A. Pierantonio. Montages-Homepage, 1997. http://univaq.it/ alfonso/montages/Mhome.html.

[BCD+87] P. Borra, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. Technical Report 777, INRIA, Sophia Antipolis, 1987.

[BD96] E. Börger and I. Durdanović. Correctness of compiling Occam to Transputer code. *Computer Journal*, 39(1):52–92, 1996.

[BDR94a] E. Börger and D. D. Rosenzweig. The WAM – Definition and Compiler Correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence, chapter 2, pages 20–90. North-Holland, 1994.

[BDR94b] E. Börger, I. Durdanović, and D. Rosenzweig. Occam: Specification and Compiler Correctness. Part I: The Primary Model. In *IFIP 13th World Computer Congress, Volume I: Technology/Foundations*, pages 489 – 508. Elsevier, Amsterdam, 1994.

[Ben86] J.L. Bentley. Programming pearls: Little languages. *Communications of the ACM*, 29(8):711–721, 1986.

[BGM95] E. Börger, U. Glässer, and W. Müller. Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines. In C. Delgado Kloos and P. T. Breuer, editors, *Formal Semantics for VHDL*, pages 107–139. Kluwer Academic Publishers, 1995.

[BR94] E. Börger and D. Rosenzweig. A Mathematical Definition of Full Prolog. In *Science of Computer Programming*, volume 24, pages 249–286. North-Holland, 1994.

[BS98] E. Börger and W. Schulte. Programmer friendly modular definition of the semantics of java. In *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998. to appear.

[CCS93] E.F. Codd, S.B. Codd, and C.T. Salley. Beyond decision support. *Computerworld*, 27(30), July 1993.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice–Hall, 1976.

[Gan85] H. Ganzinger. Modular first-order specifications of operational semantics. In H. Ganzinger and N.D. Jones, editors, *Programs as Data Objects*, number 217 in LNCS. Springer, 1985.

[GE90] J. Grosch and H. Emmelmann. A tool box for compiler construction. Technical Report 20, GMD, Universitä Karlsruhe, January 1990.

[GH93] Y. Gurevich and J.K. Huggins. *The Semantics of the C Programming Language*, volume 702 of *LNCS*, pages 274 – 308. Springer Verlag, 1993.

[GHL+92] R.W. Gray, V.P. Heuring, S.P. Levi, A.M. Sloane, and W.M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, (35):121 – 131, February 1992.

[GK97] U. Glässer and R. Karges. Abstract State Machine Semantics of SDL. *Journal of Universal Computer Science*, 3(12):1382–1414, 1997.

[Gur88] Y. Gurevich. Logic and the challenge of computer science. In E. Böerger, editor, *Current Trends in Theoretical Computer Science*, pages 1 – 57. CS Press, 1988.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods.* Oxford University Press, 1995.

[HB88] R.M. Herndon and V.A. Berzins. The realizable benefits of a language prototyping language. *IEEE Transactions on Software Engineering*, 14:803–809, 1988.

[Hen91] M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structured Operational Semantics.* Wiley, New York, 1991.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576 – 581, 1969.

[Hug] J. Huggins. Abstract state machines web page. http://www.eecs.umich.edu/gasm.

[Kli93] P. Klint. A meta–environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, April 1993.

[KP97a] P.W. Kutter and A. Pierantonio. The formal specification of Oberon. *J.UCS*, 3(5):443 – 503, 1997.

[KP97b] P.W. Kutter and A. Pierantonio. Montages: Specifications of realistic programming languages. *J.UCS*, 3(5):416 – 442, 1997.

[Kut97] P.W. Kutter. Dynamic semantics of the programming language Oberon. Technical Report 27, Computer Engineering and Networks Laboratory, ETH Zürich, 1997.

[Kut98] P.W. Kutter. An ASM Macro Language for Sets. Technical Report 34, Computer Engineering and Networks Laboratory, ETH Zürich, 1998.

[Mar97] R. Marti. Das cubix-metamodell. SBG–ORIM, 1997. Internal publication of Union Bank of Switzerland.

[Ode89] M. Odersky. *A New Approach to Formal Language Definition and its Application to Oberon.* PhD thesis, ETH Zürich, 1989.

[PDC92] T.J. Parr, H.G. Dietz, and W.E. Cohen. Pccts reference manual (version 1.00). *ACM SIGPLAN Notices*, pages 88 – 165, February 1992.

[PH91] A. Poetzsch-Heffter. *Formale Spezifikation der kontextabhängigen Syntax von Programmiersprachen.* PhD thesis, Technische Uni. München, 1991.

[PH97] A. Poetzsch-Heffter. Prototyping realistic programming languages based on formal specifications. *Acta Informatica*, 34:737–772, 1997. 1997.

[Sch86] D.A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., 1986.

[Sch97] D.A. Schmid. On the need for a popular formal semantics. *Sigplan Notices*, 1997.

[SS71] D.S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. *Computers and Automata*, (21):14 – 46, 1971. Microwave Research Institute Symposia.

[Sto77] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Wal95] C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.

[Wei97] D.M. Weiss. Creating domain–specific languages: The fast process. In S. Kamin, editor, *First ACM-SIGPLAN Workshop on Domain–Specific Languages DSL'97*, 1997.

[ZG97] W. Zimmermann and T. Gaul. On the construction of correct compiler backends: An asm approach. *Journal of Universal Computer Science*, 3(5):504 – 567, 1997.

Flexible and Reliable Process Model Properties: An Integrated Approach

Tiziana Margaria[1] and Volker Gruhn[2]

[1] METAFrame Technologies GmbH,
Joseph-von-Fraunhofer-Str. 20, D-44227 Dortmund (Germany)
tel: +49 231 9700.980, fax: +49 231 9700.989

[2] Universität Dortmund, Baroper Str.301,
D-44221 Dortmund (Germany), tel: +49 231 755.2782, fax: +49 231 755.2061,
gruhn@ls10.informatik.uni-dortmund.de

Abstract. In this article we describe an integrated approach to process management based on the widely used LEU toolset for process modelling and workflow management and on the ABC tools for formal verification of process model properties. We show how process modelling and process model analysis benefit from this integration by gaining a fully automatic global property check capability. We illustrate the approach by means of a process model example taken from an industrial project.

1 Motivation

The growing demand for systematization and support in the design and enforcement of software processes and workflows is a current issue in organizations and businesses. To be competitive, enterprises must be able to react promptly to unpredictable need shifts. This implies a capability to tailor and reshape their processes in a pervasive way and under time pressure, improving their global organization without losing reliability and operability. Vital properties of the process models (PMs), which span workflow management [11], business process (re)engineering [29], actually the whole of software process modelling [19], must thus be guaranteed.

Techniques available in the formal methods community are in principle capable of covering these demands, but typically at the price of extensive redesign and reorganization accompanying the introduction of new comprehensive methods (like [1]) which drastically limit the acceptance. In our experience, practitioners are more likely to adopt new tools rather than new methods. In particular automatic tools have the capability to encapsulate the essence of the formal techniques and make them more readily available even to untrained personnel.

A class of properties currently perceived as particularly hideous concerns loose global constraints: expressing properties of entire processes, not of isolated components. They describe desired interplays or feared interactions among seemingly independent portions of complex processes. Such properties are not yet covered by software modelling and process modelling environments, and therefore

not even formalizable in practice [5]. However, the more complex the processes become, the more global constraints arise, and the more design expertise and simulation or testing effort are required in order to master these dependencies.

In this paper we show how to attack this problem by integrating two complementary technologies, the LEU tools and the ABC tools, at the tool level. The integrated environment is a process modelling environment that combines workflow management with formal verification techniques and offers complete tool support for the design, simulation, verification of the process models and for workflow management.

Concretely, the LEU tools [12] form a process management environment which is commercially available since 1993 and which is well accepted and widely used in practice. They are used for process modelling, process simulation, and workflow management. In our integrated environment, workflow models produced by LEU can be verified via the formal techniques available within METAFrame's ABC.

The ABC has been used in different industrial settings to deal with high-level modelling, animation, (formal) verification, and synthesis of models (called services) in a largely application-independent fashion. Addressing different aspects and abstraction levels, the two toolsets cover different facets of the complete process management life cycle. Their potential for synergies is consequently exploited in the proposed integration approach which is based on an abstract semantics of PMs.

The ease of acceptance of the integrated environment depends on its immediate compatibility with today's systems, practice and habits. Both technologies are conceived for a proficient use even with a minimum level of expertise: this reduction of the initial threshold is in our experience a de-facto precondition for a wider dissemination of formal techniques. Tool builders may give a significant contribution to achieving this task, which leads us to stress the 'tool builder's perspective' in the remainder of the paper. Instead of going into the detail of the concrete realization, we rather discuss the adopted guidelines, which are dictated by the above considerations. They indeed largely influenced our design decisions, which would be in many cases different in a purely academic setting.

In the following we sum up our requirements for the integrated environment (Sect. 2) and present LEU and ABC, the pre-existing environments we build on (Sect. 3). We then sketch the idea of integration, whose several aspects are dealt with in Sect. 4. Finally, Sect. 5 contains our expectations and perspectives.

2 Requirements for the Integrated Environment

A central requirement to the integrated environment is its *acceptability* in today's industrial practice: this means a graceful insertion in today's software and business process engineering context. Small and medium size enterprises are here under particular pressure: they cannot afford experimenting, nor any loss of productivity of their teams while migrating to new technologies. New technologies must be immediately usable.

The **high-level requirements** emerging from all the meetings with customers were largely independent of the concrete application domain and techniques. Unanimously, the new environment must be:

- *compatible with already used tools and methods*, to be immediately usable with virtually no extra training. Important here are a graphical presentation layer and the availabilty of online help and user's manual.
- *oriented at the application level*, to address directly domain experts (e.g. insurance employees or telecommunication service designers involved in process modelling) with little or no understanding of the tool's interna. Any need of understanding the software is in fact perceived as requiring programming skills. Even a (maybe only initial) mediation by software experts is largely felt as intrusion in the own competence domain, thus basically undesired.
- *incrementally usable* [27], in order to introduce the use of the new features *on demand*, 'discovering' them one by one whenever a concrete situation requires more support than the current practice offers. This 'bootstrapping' phase is of crucial importance to the real acceptance of the whole environment. In our experience, positive feedback is communicated quite quickly in a team, and leads to increased openness towards the new techniques even by the initially most skeptical members.

The **functional requirements** are of more technical nature, and are discovered interactively, analyzing the current practices and their shortcomings. The general feeling was a need for *tools which support distributed, remote development and management of large, hierarchical, heterogeneous, decentral process models.* These still emerging aspects are handled largely by hand, on a case by case basis. However, designers feel the danger of losing control over the interdependencies with the increasing complexity of the processes, wish process modelling environments to be delegated the coordination, and maintain and manage their overview. The functional requirements to an improved workflow management environment are therefore quite ambitious:

- *support both reuse of components and reuse of workflows*, thus taking a coarse-grain approach
- *enable and manage 'design in the large' strategies*, keeping track of the coherency of decisions
- *actively support teamwork* by (i) taking care of different competences of team members due to different roles and professional profiles and (ii) managing the evolving knowledge, due for example to changing personnel
- *manage heterogeneous application environments*, whereby a rich collection of design tools with different application profiles is needed to allow an application-specific treatment of complex design tasks
- *follow a coordination-oriented approach* instead of proposing a high-level programming method.

The functionality offered by the integration of LEU and ABC covers all the previous requirements. The design trade-offs in the integration are consciently

resolved in first instance in favour of simplicity, to favour automation over more refined capabilities. Refinements covering particular aspects can be still done incrementally, on demand, enriching the global functionalities.

3 LEU and ABC: The Initial Situation

The LEU and the ABC tools are in the following briefly described from the point of view relevant to this paper. Focussing on different aspects, taken singularly they cover two different application areas. The integration concept shows then how to exploit the synergies for a global environment offering unique features.

3.1 The LEU Tools

The LEU tools [12] describe process models in terms of FUNSOFT nets and use these models for illustration purposes, for process simulation and its quantitative evaluation, and for workflow management.

FUNSOFT nets are abstract Petri nets which fulfill the requirements for process modelling languages. They are bipartite graphs where one sort of nodes is called agencies, the other is called channels.

Agencies represent activities. They read objects/documents from their preset channels[1] and write objects/documents to their postset channels. Agencies have the following attributes:

- the firing time specifies how much time elapses between reading of objects from the preset channels and writing of objects to the postset channels during process simulation,
- the execution mode can either be *manual* (for agencies whose execution requires human participation) or *automatic* for agencies which can automatically be executed as soon as all preset channels are filled, and
- the definition of how often an agency can be fired simultaneously.

Channels constitute object and document stores. Each channel has a name (found below the channel in the graphical representation) and is associated with an object type or a document type (above the channel symbol). E.g., channels associated with the object type *tenant* can only store objects of that type.

Figure 2 (left) shows a FUNSOFT net representing the process of creating a lease in a real-estate management system. The five agencies of this process are connected to each other via channels. An object of document type *document* is read by the agency *selection* from channel *request*, modelled through an edge between a channel and an agency. The agency *calculation* is connected with the channel *rent level* via a copying edge (graphically illustrated by the small circle at the bottom edge), thus defining that the agency reads objects from the channel without removing them. Reading edges describe a destructive reading.

[1] A channel s will be called preset channel of an agency t, if there is an edge from s to t. If there is an edge from t to s, then s will be called postset channel of t.

The agency *selection* selects a tenant, a landlord and an accommodation unit to be combined in one contract in the agency *create lease*. This agency needs also the calculated rent, supplied by agency *calculation*. The generated lease is archived and submitted to a precheck, whereby agency *precheck* reads an object of type *lease*. Depending on the precheck outcome, the lease is either accepted and written into the channel *lease to legal department*, or forwarded to the channel *request* with added notes. This either-or firing behaviour is an extension of the normal Petri net firing behaviour simplifying the modelling of decisions. There are a number of other firing behaviours in the FUNSOFT net, which have appeared to have made modelling easier in the framework of modelling business processes.

Structuring Process Models. The FUNSOFT net approach supports:

Agency refinement An agency is refined [14] by describing details of the represented activity by means of a further FUNSOFT net. For example, the symbol for the agency *precheck* in figure 2 (left) shows that details of this activity are described through a FUNSOFT net. The refined agency *precheck* on the upper level is just a placeholder for the refining net.

Binding of process models to agencies Such a binding defines that a process of the connected model will be started as soon as the agency may be activated. The binding of models to agencies ensures that frequently used processes will not have to be modelled more than once.

Interface channels To coordinate a number of processes, be they within a company or across company borders, it is necessary to define interfaces between processes. In Fig. 2 (left), the agency *precheck* writes a lease into the interface channel *lease to legal department* (represented by the symbol for interface channels). Objects written into an interface channel are passed to all the processes reading from that channel. In our example, the lease will be transferred to a process in the legal department.

Semantics of FUNSOFT Nets. The semantics of FUNSOFT nets is defined through a mapping onto Predicate/Transition nets (Pr/T nets) [10]. The mapping [9] implements a local simulation [21] of FUNSOFT nets through Pr/T nets with Many-sorted Structures, Multi-Sets and a weak transition rule [10].

Validation via Simulation. Starting from syntactically correct process models, the process model analysis identifies logical errors and inefficiency of process models. This analysis offers the earliest possible detection of errors, thus it helps avoiding the use of incorrect models for workflow management.

A process model M is validated through the simulation of processes $G_1, ..., G_n \in GP(M)$[2]. The effects of firing agencies are imitated in process simulation. This means that objects are read from preset channels and written into postset channels. The values of such objects are ignored: only object identifiers are used in the simulation. Conflicts between agencies can be influenced in the simulation by

[2] GP(M) describes the set of enactable processes on the basis of process model M.

giving details about the probability of each agency potentially sharing a conflict. Decisions within processes depending on values of objects and documents to be processed can be imitated through manual interference in the simulation (e.g. naming due decisions). The simulation of a process gives an impression about process states, without wasting the resources which are needed for a real process. The simulation of processes can be analyzed in two different ways. On the one hand, a simulation run is animated, meaning that it is possible to observe which agencies fire at what time, which channels they read from and write to, and how many objects are stored in which channels. On the other hand, basic events, like reading and writing of objects by agencies and the insurgence of conflicts when a number of agencies compete about the access to an object, are recorded during the simulation. Simulation traces can be analyzed statistically and the results can be partly visualized (critical paths, bottlenecks, degree of parallelism) [8].

Workflow Tracing and Feedback. Information about activities execution can be also recorded at runtime, during real workflow management. Workflow traces are more expressive than simulation traces, since the assumptions on which the simulation is based (e.g. about the time needed for certain activities or about stochastic conflict resolution) are replaced by the real process behaviour.
A workflow trace may e.g. contain information about the actual running time of single activities, the people taking part in the activities, about dynamic modification of process models (i.e. changes of a model during workflow management), and about the solution of conflicts.

Observed Shortcomings. Unfortunately, FUNSOFT nets provide an inadequate support for proving properties of certain process model parts and also for compositional proofs. A further drawback of the current process model analysis is its lack of support of *global constraints*: they can be neither expressed nor checked.

Global constraints express restrictions which have to be respected by the collection of all single process models. It is extremely hard to model such constraints on a local basis, since it requires otherwise unnecessary duplication of context or distributed information which has to be added and maintained for all the single process models concerned. The process model of figure 2, for example, expresses how a lease contract is agreed on. A global constraint which must obviously hold is that

> *for each potential tenant a liability check should be carried out before issueing a contract.*

Modelling this constraint on a local basis involves modifying all the process models which deal with tenants, e.g. by introducing local liability checks. Doing this for several constraints results in process models which are much more complicated than necessary. In fact, this modelling practice is in reality followed only for selected, really critical constraints, often resulting in underspecifications which lead to runtime exceptions or failures.

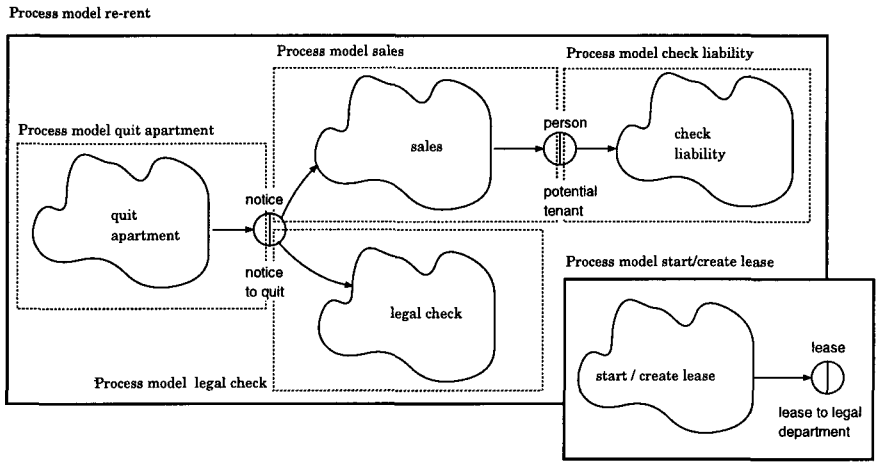


Fig. 1. Hierarchical Process Modelling in the Integrated Environment

The Solution: Model Checking the PM Graph: Being able to define and automatically check this condition as a global constraint for all the PMs dealing with tenants is a real improvement over the current state-of-the-art. Individual process models can then remain simple and comprehensible if we are able to model a global level of workflows.

This is illustrated in figure 1, where we recall a concrete example from [6], where a set of single process models was related by interface channels. In [6], channel *notice to quit*, for example, served as interface between process models *quit apartment* and *sales*, but the modelling did not make it explicit: designers must know themselves the dependencies and links among large sets of single PMs, like e.g. which activities, spread over several processes, may exchange information. In contrast, here we directly see that the four process models - despite having been modelled independently - do interact with each other!

In figure 1, PM *start/create lease* with interface channel *lease to legal department* is for the moment still isolated: this interface channel is not statically linked to any other process model. In this situation it might be difficult to detect whether the liability of a tenant is checked at the right time. Here is where model checking based on ABC helps to keep things simple: we can in fact 'look' at this process model in the context of the other one, express global constraints, and verify them automatically via the model checking component of ABC.

3.2 The ABC Tools

ABC is a coarse-grain software development environment with focus on coordination, verification, and synthesis [25] by means of fully automatic formal methods. It has been already successfully used for applications as diverse as IN SD [4, 26] or personalized E-Commerce [18], and it currently supports the ETI online Interactive Tool Coordination platform [25].

Central characteristic is its support of reasoning in presence of abstraction. Working in a model-oriented way, it allows capturing global interplays of even large systems at a tailorable granularity with a focus on verification of properties and model synthesis. In the context of the LEU features, the ABC offers:

1. **Modelling and checking of global workflows**: Global properties of control flows are expressed as temporal and modal logic formulas. The environment offers
   - automatic verification of branching time properties (expressed internally in mu-calculus [15]) for finite and infinite models by means of the Fixpoint Analysis Machine [23].
   - automatic synthesis of workflows satisfying linear time formulas [25],
   - support of hierarchy in a fashion compatible with LEU's refinement.
2. **Dealing with objects and organization**: the ABC Dataflow Analysis component offers a set of intra- and interprocedural analyses implemented themselves via model checking. This complements the LEU analysis capabilities of FUNSOFT Nets.
3. **Tailoring the Presentation Layer**: The look and feel of the presentation layer is also supported by formal methods: abstract views and error views are obtained via abstract interpretation and property-driven model collapse [26]. The application-specific and even user-specific customization of those views is also achieved via the specification and realization of special views on the basis of formal methods [17].

3.3 The Integration Approach

The key idea for the integration of both environments is a property-oriented treatment of PMs at the semantic level along a *unifying models* approach [22]. Here we motivate this choice, while the concrete realization is described in the next section.

In a unifying models setting, the operational, model-based approach has several advantages: it is fairly intuitive, thus simple, and it allows a flexible balance between expressive power and manageability which can be exploited for tailoring specific analyses.

Given a system whose aspects are described according to different paradigms or viewpoints, the underlying models must be powerful enough to capture the interference potential between the different aspects of the descriptions methods, while remaining simple enough to support (automatic) verification. If this holds, we are able to reconcile different aspects like the expression and check of control constraints or of abstract liveness and safety properties, but also architectural, network or process descriptions, or the analysis of communication or synchronization structures.

The first requirement, however, is consistency. The notion of consistency is intuitive in an operational setting: two (aspects of) descriptions are consistent if there exists a concrete system whose operational behaviour satisfies the requirements of both of them. Additionally, operational methods enjoy a mature

and elaborate *tool support*: powerful decision tools based on model checking and model construction algorithms offer even full automation, and, in case of failure detection, they may even provide intuitive diagnostic information for its cause. These tools can be applied in particular to decide consistency, thus dramatically reducing the required expertise for a competent use of multi-paradigm description methods, as is desired in our case.

4 Integrating LEU and the ABC

For the current integration task we built on previous experiences, like the definition of a multi-paradigm approach for requirements engineering proposed in [3]. There, a constraint language (SLTL [25]) and a hierarchical state-based language (Statecharts [13]) have been used in combination, in order to consistently cover global constraints and structural PM properties. A similar experience has been made in the software process modelling community where it has been identified that multi-paradigm approaches are needed to express all requirements for process models appropriately [7, 16]. In the present setting,

- temporal and modal logics, provided by the ABC, express abstract specifications in the form of loose global constraints [25], like ordering requirements, or abstract safety and liveness properties, whereas
- FUNSOFT nets, the central models in LEU, support the development of detailed process models at the concrete level.

The link between these two specification layers is established within an operational framework, coherently with the unifying models approach.

The Models. In the basic setting considered in this first phase, the unifying models are formally realized in terms of temporal model structures, which can be regarded as combinations of Kripke structures and finite state transition systems [28]. In fact, combined with adequate abstraction, refinement, and (non-standard) interpretation, this delivers an intuitive and flexible framework: varying the granularity or the concept of observation, i.e. of what is considered a visible 'action' of the system, these structures are able to cover a wide range of consistency aspects even for distributed systems.

Instead of considering the bottom level granularity of FUNSOFT Nets (Pr/T nets), in the ABC modelling we take a coarse-grain approach: as shown in Fig. 2, we interpret the FUNSOFT Nets as ABC Service Logic Graphs, in which both FUNSOFT activities and channels are (labelled) nodes and the edges maintain their labels. Thanks to ABC's polymorphic labels [2], we can bind labels to the graph elements containing different kinds of information. In the most abstract case it might be just the name of the activity/channel or edge, but it may be enriched at need with attributes, even simulation and execution code.

Depending on the required analysis, the resulting ABC graphs can be interpreted as Labelled Transition Systems, Kripke structures, or program structures,

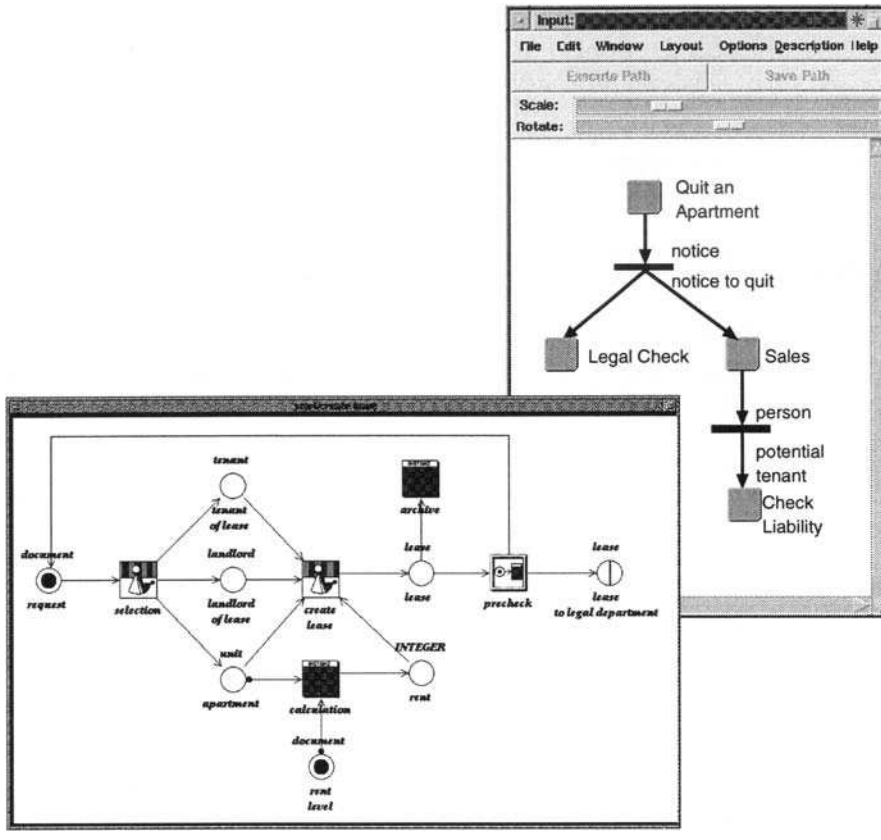


Fig. 2. The 'Lease' PM Example in the Integrated Environment

thus allowing the immediate application of the rich set of analyses and verification methods available in the ABC. Refinement of FUNSOFT nets corresponds to macro expansion, which is available in the ABC and compatible with its verification algorithms.

In addition, we may use the ABC hierarchical modelling facility to draw global PMs, like in Fig. 2, which concerns the same processes and interdependencies schematically depicted in Fig. 1. The ABC window (right) shows the composed process **re-rent**, the LEU window (left) shows the atomic process **start/create lease**.

Verification. In accordance to the requirements of Sect. 2, for the first phase of our project we have concentrated on the integration of the LEU process modelling features with the ABC automatic verification algorithms. They concern

- a **semi-automatic synthesis procedure**, where sequential portions of PMs, automatically synthesized from abstract specifications, can be manually composed into structured FUNSOFT nets, and

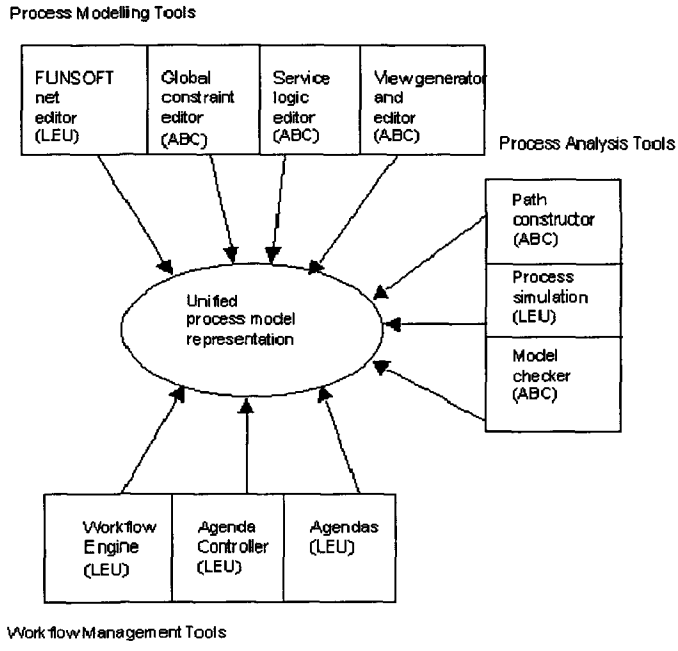


Fig. 3. Architecture of the Integrated Environment

- automatic formal verification via model checking, which validates the global constraints for the resulting PM descriptions.

This way, we support both a top-down, refinement based development style, and a bottom-up style which is more closely compatible to the current practice.

4.1 The Integration Architecture

The integration of the existing tools takes place at the functionality level. It is therefore difficult to sketch an architecture of the combined software environment from the classical software point of view. It is much more informative to show how the different functionalities interact and complete each other.

The architecture shown in figure 3 shows that we distinguish three types of components: process modelling components, process analysis components and workflow management components. These components are either original ABC or original LEU components. All components are integrated over a unifying process model representation. This representation can be computed at need, depending on the specific task, in order to avoid the explosion of information which may arise in general purpose common formats. Thus it contains at least the information needed about the modelled processes to perform the current specific task. This unifying representation philosophy is used both for process analysis purposes and for workflow management purposes. Thus, we ensure by means

of several task specific abstractions and views that all the model information is considered in process model analysis and in workflow management.

4.2 Realization and Examples

In the present scenario, we must be able to deal with process models at different levels of concretization, not only to deal with compositionality, but also in order to support different viewpoints on the system.

Whereas the need for abstraction and refinement for adapting descriptions of different granularity is quite clear, *non-standard interpretations* of the model structures are the key for obtaining a satisfactory framework for consistency checking in the presence of true *heterogeneity*: establishing a match between the imposed constraints may required to interpret the same model structure differently in different contexts. This goes beyond standard semantic approaches that merely abstract from the representations chosen for the various description methods. Non standard interpretations play a role at the local as well as at the global level, as exemplified in the following.

Multidimensional descriptions do not only cover the desired functionalities, but also orthogonal dimensions like the organization schema of a project or a company, the deployment of resources or the location of team members. These facets are captured in taxonomic descriptions [25] which visualize the dependencies within and among the facets.

In the concrete example, the activity precheck in Fig. 2 is associated to its refinement in terms of a further FUNSOFT net (like in the usual refinement semantics of LEU), but also to a very abstract characterization in terms of *place of execution* (say, *Dortmund*), operator (say *agent, employee, administrative, Mr. Müller*), execution modality (*manual*) and approximate duration (*30 min*).

Global constraints express the interdependencies along complex PMs, capturing in a declarative thus simple form

- *control, validity, applicability constraints*, like
  The tenant's liability check must be performed before submitting a location contract, formalized in user-level SLTL as

  ```
  check liability before lease
  ```

- *architectural, network or process descriptions* explicitly captured in the high-level process modelling of Fig. 2, in which each node of the graph represents a FUNSOFT net, i.e. a process model, itself
- *communication or synchronization patterns* among and across process models, also clarified in the high-level process modelling of Fig. 2 by means of clear interfaces between the component models
- *abstract liveness and safety properties* specifying at a higher abstraction level the correctness of the modelling, like, in user-level SLTL

  ```
  (lease or renew-lease) and finally (quit or renew-lease)
  ```

depending on the aspect or view of the system currently under consideration.

The local aspects (here activity and channel names, like *lease*, *check liability*) are incorporated in the global constraints via an extension of propositions and actions to range over predicate logic formulas over the conjunction of the taxonomies (like in *lease* $\vee$ *renew-lease*).

Application Scenarios The power of this approach becomes clear when picturing it in conjunction to non-standard interpretations: unifying models may be

1. **an architectural view of the system**, where the local aspects concern the hardware/software composition and configuration of some computation nodes of a distributed system, and where global properties express compatibility constraints over the whole topology
2. **an organizational view of the agents**, including the people directly involved in parts of the process and the software agents which carry over some dedicated tasks, together maybe with the people responsible for their update and maintenance. The local description concerns activities, responsibilities, and the organizational chart of the company/companies they belong to or pointers to it, and the global descriptions characterize consistent configurations in which the team can complete the tasks, possible substitutions and online reconfigurations as well as predict the degradation of the overall service/performance.
   a functional view of the workflows, in which the specified properties are the tasks to accomplish, the available components are single actions to be taken or software modules to be run, and the single tasks must themselves be coordinated at a higher abstraction level.

These views are less distant to each other than it may initially appear: they coexist in a large application to real-estate management, in which the whole administration, renting, maintenance, and customer care for a large number of apartments distributed over a large territory has to be coordinated.

5 Expectations and Perspectives

The proposed integration of LEU and ABC conjoins the strengths of both toolsets into an environment with increased functionality and low acceptance threshold.

The process modelling facilities of LEU build the interface to the process modeller, so that application designers familiar with FUNSOFT nets and their simulation are not bothered by new formalisms or user interfaces. This perfectly matches with the high-level requirements to the enviroment (compatibility with already used tools, orientation at the application level, incremental usage).

The substantial functional shortcomings of LEU (inadequate support for proving certain types of properties and definition of global constraints) are overcome by resorting to adequate ABC components. Full automation of the used

formal methods and the intuitive formulation language for the global constraints account for an increased functionality with little or no impact on the acceptance.

We believe that the improved modelling comfort (with respect to global process models and constraints) and the extended verification capabilities will be adopted by most process modellers used to LEU. Although the functionality offered by the integration of LEU and ABC is for the moment perceived as an add-on, it may become a must in the near future for all kinds of process management environments.

The proposed consistency modelling and checking is only a first step towards an integrated, tool-controlled process management framework which flexibly covers various paradigms. Nevertheless, we are convinced that the current proposal is a major step towards reliable and consistent system development with a strong practical impact. Like type checking, which dramatically increased reliability of programming while being (even in its intent) far from full verification, using a consistency model on-demand together with adequate tool support can efficiently help application designers and engineers to automatically detect misconceptions already in early design phases.

The availability of low-threshold, automatic methods is in our opinion a viable strategy to achieve a wide dissemination of formal methods [24] in an industrial context. Once the use of 'lightweight', fully automatic approaches starts being generally perceived as a benefit, the demand for more powerful, but (semi-)interactive formal methods, like e.g. theorem proving techniques, will gain a much stronger standing and support in the application community.

References

1. J.R. Abrial *The B Book, Assigning Programs to Meaning*, Cambr. Univ. Press, 1996
2. M.v.d. Beeck, V. Braun, A. Claßen et al.: *Graphs in* METAFrame*: The Unifying Power of Polymorphism*, Proc. TACAS'97, Enschede (NL), LNCS 1217, Springer Verlag, April 1997.
3. M. v.d. Beeck, T. Margaria, B. Steffen: *A Formal Requirements Engineering Method Combining Specification, Synthesis, and Verification*, Proc. IEEE SEE'97, Cottbus (D), Apr. 1997, IEEE Comp. Soc. Press, pp.131-144.
4. V. Braun, T. Margaria, B. Steffen, F.-K. Bruhns: *Service Definition for Intelligent Networks: Experience in a Leading-edge Technological Project Based on Constraint Techniques*, Proc. PACT'97, 3rd Int. Conf. on Practical Applic. of Constraint Technology, April 1997, London (UK), Ed. by The Practical Application Company,
5. W. Deiters and V. Gruhn. *Software Process Model Analysis Based on FUNSOFT Nets. Mathematical Modeling and Simulation*, (8), May 1991.
6. W. Deiters and V. Gruhn. *Process Management in Practice. Applying the FUNSOFT Net Approach to Large-Scale Processes. Automated Software Engineering*, (5), pp. 7-25, Kluwer, 1991.
7. W. Deiters, V. Gruhn, W. Schäfer. *Process Programming: A structured Multi-Paradigm Approach Could be Achieved.* In *Proc.* 5^{th} *Int. Software Process Workshop*, Kennebunkport (USA), Sept. 1989.
8. G. Dinkhoff, V. Gruhn, A. Saalmann, M. Zielonka. *Business Process Modeling in the Workflow Management Environment LEU. Proc. 13. Int. Conf. on the Entity-Relationship Approach*, pp.46-63, Manchester (UK), Dec.1994, Springer, LNCS 881.

9. W. Emmerich, V. Gruhn. *FUNSOFT Nets: A Petri-Net based Software Process Modeling Language.* Proc. 6^{th} Int. Worksh. on Software Specification and Design, pp.175–184, Como (I), Sept. 1991.
10. H.J. Genrich. *Predicate/Transition Nets.* In *Petri Nets: Central Models and Their Properties*, pp.208-247, Berlin (D), 1987. Springer, LNCS N. 254.
11. D. Georgakopoulos, M.F. Hornick, A. Sheth. *An Overview of Workflow Management: from Process Modeling to Workflow Automation Infrastructure. Journ. of Distributed and Parallel Databases*, 3(2), 1995.
12. V. Gruhn. *Business Process Modeling and Workflow Management. Intelligent and Cooperative Information Systems*, 4(2):145–164, 1995.
13. D. Harel: *Statecharts: A visual formalism for complex systems*, Science of Computer Programming, vol. 8, no.3, pp.231-274, 1987
14. P. Huber, K. Jensen, R.M. Shapiro. *Hierarchies in Coloured Petri Nets.* Proc. 10^{th} Int. Conf. on Application and Theory of Petri Nets, pp. 192-209, Bonn (D), 1989.
15. D. Kozen: "*Results on the Propositional μ-Calculus*", Theoretical Computer Science, Vol. 27, 1983, pp. 333-354.
16. U. Leonhardt, J. Kramer, B. Nuseibeh, A. Finkelstein. *Decentralised Process Modelling in a Multi-Perspective Development Environment.* Proc. ICSE 17, pp.255-264, Seattle (USA), April 1995.
17. T. Margaria, V. Braun: *Formal Methods and Customized Visualization: A Fruitful Symbiosis* In *Services and Visualization*, LNCS 1385, Springer Verlag.
18. T. Margaria, V. Braun, B. Steffen, R. Rückert: *Personalized Electronic Commerce Services*, to appear in Annual Review of Communication, Int. Engineering Consortium Chicago (USA), IEC, 1998.
19. C. Montangero, ed. *Software Process Technology - Proc. 5^{th} Europ. Worksh. on Software Process Modelling*, Nancy (F), Oct. 1996, Springer, LNCS N. 1149.
20. G. Plotkin: *A Structural Approach to Operational Semantics*, Univ. of Aarhus (DK), DAIMI FN-19, 1981.
21. P. Starke. *On the mutual simulatability of different types of Petri nets.* In *Concurrency and Nets*, pp. 481–495, Berlin, FRG, 1987. Springer.
22. B. Steffen: *Unifying Models*, Inv. talk, 14th Symp. on Theoretical Aspects of Computer Science (STACS'97), Feb.-Mar. 1997, Lübeck (D), LNCS, Springer Verlag.
23. B. Steffen, A. Claßen, M. Klein, J. Knoop, T. Margaria: *The Fixpoint Analysis Machine*, Proc. CONCUR'95, Pittsburgh (USA), Aug. 1995, Springer, LNCS 962.
24. B. Steffen, T. Margaria: *Tools Get Formal Methods into Practice*, pos. statement, ACM Worksh. on *Strategic Directions in Computing Research*, ACM Computing Surveys 28A(4), Dec. 1996, `http://www.acm.org/surveys/1996/SteffenTools/`
25. B. Steffen, T. Margaria, V. Braun: *The Electronic Tool Integration Platform: Concepts and Design*, Int. Jour. on Software Tools for Technology Transfer, Vol.1, Springer Verlag, Dec. 1997 , pp. 9-31.
26. B. Steffen, T. Margaria, V. Braun, N. Kalt: *Hierarchical Service Definition*, Annual Review of Communication, Int. Eng. Consortium, Chicago (USA), IEC, 1997.
27. B. Steffen, T. Margaria, A. Claßen, V. Braun: "*Incremental Formalization: a Key to Industrial Success*", Software: Concepts and Tools 17(2), pp.78-91, Springer Verlag., July 1996.
28. C. Stirling: *Modal and Temporal Logics*, In *Handbook of Logics in Computer Science*, Vol. 2, pp. 478 – 551, Oxford Univ. Press, 1995.
29. B. Warboys. *Reflections on the Relationship Between BPR and Software Process Modelling. Proc.* 13. *Int. Conf. on the Entity-Relationship Approach*, pp.1-9, Manchester (UK) Dec. 1994. Springer, LNCS 881.

A Symbolic Model Checker for ACTL*

A. Fantechi[1,2], S. Gnesi[2], F. Mazzanti[2], R. Pugliese[1] and E. Tronci[3]

[1] Dip. di Sistemi e Informatica, Univ. di Firenze, Italy
[2] Istituto di Elaborazione dell'Informazione, C.N.R. Pisa, Italy
[3] Dip. di Matematica Applicata, Univ. dell'Aquila, Italy

Abstract. We present SAM, a symbolic model checker for ACTL, the action-based version of CTL. SAM relies on implicit representations of Labeled Transition Systems (LTSs), the semantic domain for ACTL formulae, and uses symbolic manipulation algorithms. SAM has been realized by translating (networks of) LTSs and, possibly recursive, ACTL formulae into BSP (Boolean Symbolic Programming), a programming language aiming at defining computations on boolean functions, and by using the BSP interpreter to carry out computations (i.e. verifications).

1 Introduction

The increasing reliance of many aspects of human society on highly complex computer systems requires the adoption of innovative validation techniques. In the validation of software difficulties arise from the discontinuous nature of the software behaviour. This behaviour is based on sequences of discrete transitions, with such a high number of possible evolution paths and failure modes, that exhaustive testing becomes impossible. Moreover, testing can provide information only on the tested paths. Hence, due to the lack of continuity, we cannot infer the behaviour of untested sequences from that of the tested ones.

Formal methods are mathematically based techniques that can offer a rigorous and effective way to model, design and analyze computer systems. It is increasingly accepted that the adoption of formal methods in the life cycle development of embedded systems would guarantee higher levels of dependability. It appears that, due to the lower costs of training and innovation, industries are more keen to accept formal validation techniques assessing the quality attributes of their products, obtained by a traditional life cycle, rather than a fully formal life cycle development. However, to achieve an efficient use of formal methods in industry, such methods need to be better integrated with traditional software engineering. Formal "validation" and "verification" techniques and automated support tools need to be improved so that they could be easily used by non-expert staff.

Model checking techniques [8] has been defined to verify system properties, expressed as temporal logic formulae, on finite state models of the behavior

* This work was partly supported by the ESPRIT project GUARDS and by Progetto Coordinato CNR "Metodologie e strumenti di analisi, verifica e validazione per sistemi software affidabili".

of systems. Once a model of a system has been generated, the properties are automatically verified by model checking tools and therefore these kind of tools can be easily used also by non-expert users. This in general is not true for verification techniques based on theorem proving approaches [5]. In this case, the system state is modeled in terms of set-theoretical structures, and operations are modeled by specifying their pre- and post-conditions in terms of the system state. Properties are described by invariants that must be proved to hold through the system execution by means of a theorem prover, usually with the help of the user. Due to the above reasons model checking has been preferred in industries, especially for formal verification of hardware components.

Many "prototipal" verification environments are currently available which can be used to automatically verify behavioural and logical properties of reactive and concurrent systems specified by means of process algebrae. Most of these environments (e.g. [9, 10, 17, 11, 4]) use finite state systems to model the systems under investigation and formulae of temporal logics to express properties [19, 26]. Usually, given a system, a so called "generation phase", based on the operational semantics of the language, allows the corresponding LTS to be derived. When real world applications are considered a main problem arises due to their extremely large state-spaces.

To cope with the "state-explosion" in the model generation phase some work has been recently done for the logic CTL [14], a branching-time temporal logic whose interpretation domains are Kripke structures. Indeed, the SMV model checker has been developed [7], which uses symbolic manipulation algorithms to check the satisfiability of CTL formulae. In SMV the transition relations are represented implicitly by means of boolean formulae and are implemented by means of Binary Decision Diagrams (BDDs, [6]). This usually results in a much smaller representation for the systems' transition relations thus allowing the maximum size of the systems that can be dealt with to be significantly enlarged.

CTL formulae allow properties of systems to be expressed in terms of their states. In case of systems which are described in terms of actions and state changes, such as concurrent (e.g. control) systems, it is more natural to use an action-based temporal logics to express their properties. For instance, one can use ACTL [13], the action-based version of CTL. Our effort has then been to build efficient model checking tools for action-based logics by directly relying on implicit BDD-based descriptions of systems' state spaces. In particular, we have built SAM, a symbolic model checker for μ-ACTL that relies on implicit representations of LTSs and symbolic manipulation algorithms. As logic language we have used μ-ACTL [16] since ACTL, although quite expressive (e.g. it permits to express safety and liveness properties, as well as certain "cyclic" properties), lacks of a "real" fixpoint operator which permits to express general recursive properties (whereas this is possible by using, e.g., the μ-calculus [20]). The symbolic model checker has been realized by translating (networks of) LTSs and μ-ACTL formulae into Boolean Symbolic Programming (BSP, [27]), a programming language aiming at defining computations on boolean functions, and by using the BSP interpreter to carry out computations (i.e. verifications).

The integration of SAM in JACK [4], an environment for the specification and formal verification of concurrent systems that also includes a model checker for ACTL using explicit state space representation, is in progress. A number of formal validation projects using SAM are under progress too. In [24] model checking of a hydroelectric power plant with (potentially) 10^{52} states has been carried out.

2 Background

2.1 JACK

Our starting point has been JACK (*Just another Concurrency Kit*) [4], that so far is the only verification environment including an ACTL model checker (AMC, [18]). JACK is an environment based on the use of process algebrae, automata and temporal logic formalisms, which supports many phases of the system development process. The idea behind the JACK environment is to integrate different specification and verification tools, independently developed at different research institutes (I.E.I.- C.N.R. and the University of Rome "La Sapienza" in Italy, and INRIA in France), to provide an environment in which a user can choose from several verification tools by means of a user-friendly graphic interface.

The FC2 format [21], i.e. the common representation format for data, makes it possible to exchange information among the tools integrated in JACK and to easily add other tools to the JACK environment, thus extending its potential. The FC2 format allows a Labeled Transition System (i.e. an automaton) to be represented by means of a set of tables that keep the information about state names, arc labels, and transition relations between states. The format allows nets of automata to be represented as well.

Some of the tools in JACK allow a process specification to be built. This can be done both by entering a specification in a textual form (i.e. a process algebraic term) by using MAUTO, or by drawing the automaton that describes the behavior of the process by using ATG [25]. Moreover, sophisticated graphical procedures, provided by ATG, allow a specification to be built as a network of processes (or networks). Hence, a hierarchical approach in the specification activity is also possible.

Once the specification of a system has been written, JACK permits the construction of the automaton corresponding to the behaviour of the overall system, by using either MAUTO or FC2LINK and HOGGAR (which is a BDD-based tool); this is the *model* of the system. Moreover, by using MAUTO or HOGGAR, automata can be minimized with respect to various (bisimulation) equivalences. ACTL can be used to describe temporal properties and *model checking* can be performed, by using AMC, to check whether systems (i.e. their models) satisfy the properties.

JACK has been successfully used in several case studies. In [12] JACK was used to formally specify the hardware components of a buffer system, and to

verify the correctness of the specification with respect to some safety requirements. In [3] the verification of an interlocking safety critical system developed by Ansaldo Trasporti was presented.

Model checking by using JACK has a major limiting factor, namely the state space explosion problem. Indeed, AMC can perform model checking only onto a single automaton (i.e. AMC cannot take a network of automata as a model); thus it is always necessary to generate the global automaton of the system. SAM, the extension of JACK presented in this paper, is aimed at solving the state space explosion problem.

2.2 CCS/Meije

Process algebrae [22] are generally recognized as a convenient tool for describing reactive systems at different levels of abstraction. They rely on a small set of basic operators, used to build complex descriptions from more elementary ones, and on behavioral equivalences (e.g. bisimulation) or preorders (e.g. testing), used to study the relationships between descriptions of the same system at different levels of abstraction (e.g., specification and implementation).

In the JACK environment, the process algebra used to define processes is CCS/Meije [1]. The syntax of the language is based on a set of elementary and uninterpreted actions that processes can perform and on a set of operators that permit to build complex processes from simpler ones. The syntax permits a two-layered design of *process terms*. The first level is related to *sequential regular terms*, the second one to *networks* of parallel sub-processes supporting communication and action renaming or restriction. The two-layered structure of CCS/Meije descriptions is reflected also by the graphical methodology that can be used in connection with ATG and that will be illustrated in Section 3.2.

For the syntax and the operational semantics of CCS/Meije, we refer the interested reader to [1].

2.3 μ-ACTL

In this section we briefly present the syntax of μ-ACTL [16], an extension of ACTL [13], the temporal logic used in the JACK environment, with a fixpoint operator. For the semantics of formulae, we refer the interested reader to [13] and [16].

μ-ACTL is suitable to express properties of reactive systems whose behaviour is characterized by the actions they perform and whose semantics is defined by means of LTS's. The logic can be used to define both *liveness* (something good eventually happen) and *safety* (nothing bad can happen) properties of reactive systems. Moreover, μ-ACTL is *adequate* with respect to strong bisimulation equivalence, namely if $p \sim q$, then p and q satisfy the same set of μ-ACTL formulae.

The definition of μ-ACTL relies on an auxiliary logic of action. The collection $\mathcal{AF}$ (ranged over by χ) of *action formulae* over a set of (visible) actions *Act*

(ranged over by α) is defined by the following grammar:

$$\chi ::= \alpha | \neg\chi | \chi \wedge \chi.$$

We write ff for $\alpha_0 \wedge \neg\alpha_0$, where a_0 is some chosen action, tt for $\neg ff$ and $\chi \vee \chi'$ for $\neg(\neg\chi \wedge \neg\chi')$. Intuitively, action formulae express sets of (observable) actions. Given an action formula χ, the set of the actions satisfying χ is $\kappa(\chi) = \{\alpha | \alpha \models \chi\}$.

The syntax of μ-ACTL formulae is defined by the following grammar:

$$\phi ::= tt \mid \phi \wedge \phi \mid \neg\phi \mid E\pi \mid A\pi \mid \mu Y.\phi(Y) \mid Y$$

$$\pi ::= X_\chi \phi \mid X_\tau \phi \mid \phi \, {}_\chi U \, \phi \mid \phi \, {}_\chi U_{\chi'} \, \phi$$

where Y ranges over a set Var of variables, *state formulae* are ranged over by ϕ, *path formulae* are ranged over by γ, E and A are *path quantifiers*, X and U are the *next* and *until* operators indexed by action formulae.

Several useful derived modalities can be defined, starting from the basic ones. In particular, we will write:

- $< \chi > \phi$ for $E[tt \, {}_{ff} U_\chi \phi]$ and $[\chi]\phi$ for $\neg < \chi > \neg\phi$.
- $EF\phi$ for $E[tt \, {}_{tt} U \, \phi]$, and $AF\phi$ for $A[tt \, {}_{tt} U \, \phi]$; these are called the *eventually* operators.
- $EG\phi$ for $\neg AF\neg\phi$, and $AG\phi$ for $\neg EF\neg\phi$; these are called the *always* operators.
- $\nu Y.\phi(Y)$ for $\neg\mu Y.\neg\phi(\neg Y)$; ν is called the maximal fixpoint operator.

2.4 Boolean Symbolic Programming (BSP)

BSP is a programming language aimed at defining, possibly fixpoint, computations on boolean functions [27]. BSP primitives include boolean operations, quantifiers, arithmetic operations (ALU-like). Defining processes just using boolean operations is a tedious and error prone task. Thus BSP has primitives to define processes as well. The output of a BSP program is formed by "answers" to BSP queries. Essentially BSP queries correspond to the usual queries on BDDs. E.g.: "Is a given boolean function identically 1 (true)?", "Print the set of satisfying assignments of a given boolean function". Shortly, BSP is a programming language to define symbolic computations at a logical level rather than at the BDD level.

The BSP compiler translates a BSP program into a sequence of calls to BDD primitives (essentially `if_then_else` and `compose`). This translation is done in time $O(s)$ where s is the size of the BSP program being translated. During such translation some optimization is also carried out (e.g. when possible BDD calls are moved outside of loops computing fixpoints, BDD calls are rearranged in an attempt to free BDDs as soon as possible). BSP tries to efficiently execute a given symbolic program. However, as for any programming language, it is the user responsibility to write efficient (symbolic) programs.

Implementations as well as specifications can both be defined using BSP. E.g.: a netlist of size n can be translated into a BSP program of size $O(n)$; a μ-calculus formula of size n can be translated into a BSP program of size

$O(n)$ [27]. These features make BSP suitable as a low level language to define finite state verification tasks. Moreover having implementation and specification defined using the same language enables the use of rewriting techniques to speed up verification [27]. Note that BSP is a programming language rather than a Model Checker. Thus BSP can be used for other purposes as well (e.g. see [28]).

3 The extended version of JACK

In this section we comment on the architecture of the extended version of JACK, which is depicted in Figure 1, by especially pointing out the new features introduced with respect to JACK. Then, by means of a simple example, we will show how the new environment can be used as a support for the specification and verification of reactive systems.

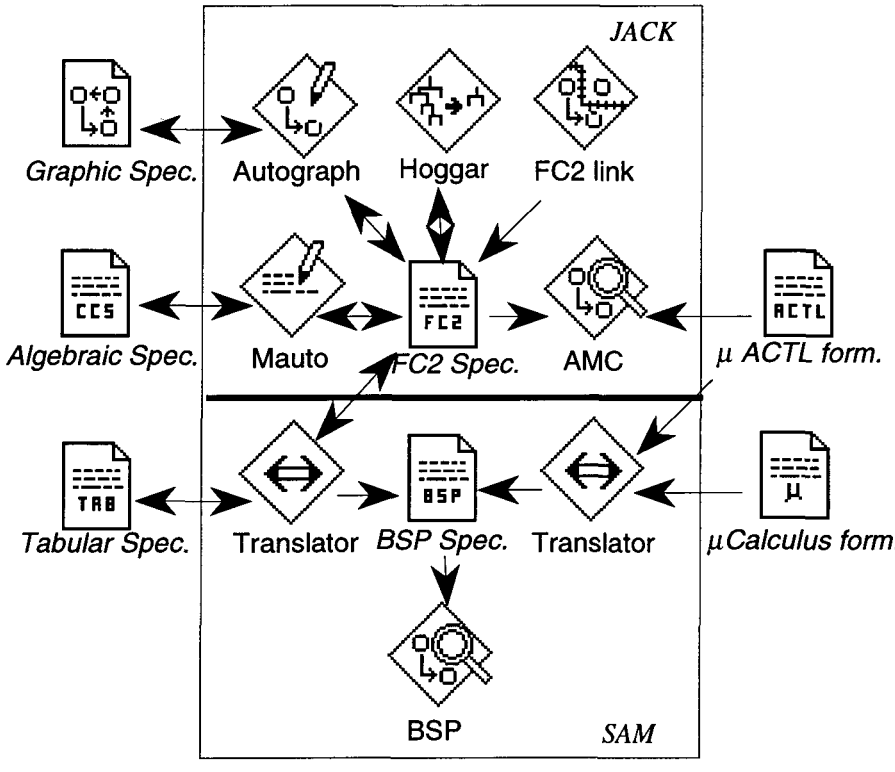


Fig. 1. The architecture of the extended version of JACK

3.1 The architecture

The user can use three different formalisms for defining the reactive systems used as models for checking logical properties. A part from textual (CCS/Meije) and graphical (ATG) representations, that both are translated first into the FC2 format and then into BSP, a *tabular* representation can be used as well, that is directly translated into BSP.

Tabular representations have the same overall structure of FC2 specifications (i.e. the description of a system is composed of the descriptions of the system components and of the descriptions of their synchronizations), but have simplified syntax and semantics so that the structure of systems is described in a more straightforward way. The tabular representation aims at helping the user to better understand system descriptions as well as to make the user able to generate by himself system descriptions directly without using graphical representations.

The user can express logical properties that must be checked as μ-calculus and μ-ACTL formulae.

Both the formalisms for specifying processes, basically FC2 and tabular representations, and the formalisms for specifying properties, namely μ-calculus and μ-ACTL, are translated, in linear time complexity, into BSP. Therefore, checking whether a reactive system s verifies a property p consists in querying the BSP interpreter if the boolean function "tr(s) implies tr(p)" is identically one, where tr(s) is the BSP program that represents s and tr(p) is the BSP program that represents p.

3.2 Symbolic model checking in practice

System specification

Let us consider a level crossing that can be crossed at a given time either by a train or by at most two cars. A train asks for the permission to cross by using action `approaching_t` and signals that it has crossed by using action `leaving_t`. A car behaves similarly but uses actions `approaching_c` and `leaving_c`, respectively. These are the only visible actions of the system. Normally, the barriers are kept open, thus cars can cross. The railway signal allows a train to proceed only if the barriers can go down safely, namely when no car is crossing. After a train has crossed, the barriers will go up again.

The specification of the system is a net, called **crossing**, composed of two automata, **barrier** representing the controller of the barrier and **signal** representing the controller of the railway signal. Figure 2 shows their graphical ATG representations. While the two component automata (top of Figure 2) should be self-explicative, here we comment on the network that describes the overall system (bottom of Figure 2). Boxes (e.g. **barrier** and **signal**) can be processes or networks, thus allowing a top-down approach in the specification activity. Ports at the border of boxes are their interconnection places. If two boxes are drawn at the same level, they can synchronize via the actions that label linked ports. In addition to CCS/Meije, a multiway synchronization operator, called *wedge*, is

available: more than two processes can synchronize by executing an action (e.g **barrier** and **signal** synchronize on `approaching_t` which labels a wedge). As in the case of a textual specification, the behaviour of a graphical specification can be defined in terms of an LTS.

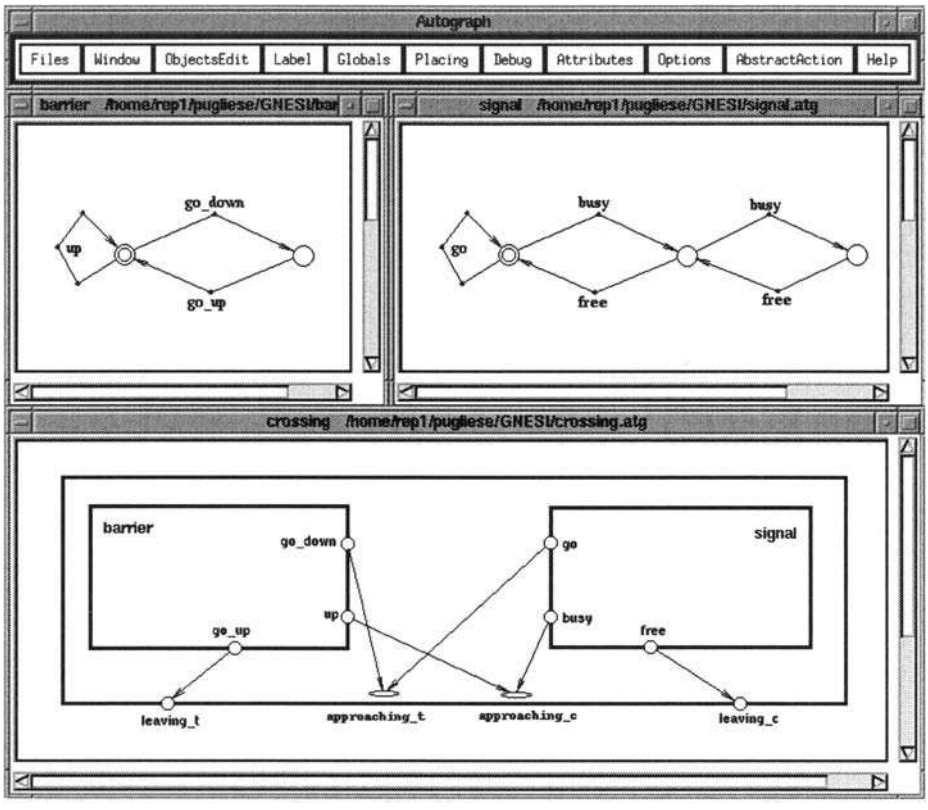


Fig. 2. The level crossing: a pictorial representation

Once the FC2 representation of **crossing** has been obtained by using ATG and FC2LINK, we can construct the corresponding tabular representation by using the utility `totab`.

This construction, shown in Figure 3, is not needed and has been done only to give a flavor of what the tabular representation is. In this representation, all the possible actions that an automaton or a net can do are explicitly enumerated. The tabular description of an automaton, other than the possible actions, specifies the possible transitions that the automaton can do. In defining the transitions, action indexes are used in place of action names. More specifically, transitions are specified as triples of numbers of the form: $n0$: $n1$ -> $n2$. Such a triple indicates that the automaton can evolve from state $n1$ to state $n2$ by performing action $n0$. The tabular description of a net with n components specifies

```
Networks: 3
Main: 1

System: 1 "crossing"        Automaton: 2 "barrier"      Automaton: 3 "signal"
Actions: 4                  Actions: 3                  Actions: 3
 1: approaching_t            1: go_down                  1: go
 2: approaching_c            2: up                       2: busy
 3: leaving_c                3: go_up                    3: free
 4: leaving_t               States: 2                   States: 3
Components: 2 3             Initial: 1                  Initial: 1
Synchronizations:           Transitions:                Transitions:
 1: 1 1                      2: 1 -> 1                   1: 1 -> 1
 2: 2 2                      1: 1 -> 2                   2: 1 -> 2
 3: 0 3                      3: 2 -> 1                   2: 2 -> 3
 4: 3 0                                                  3: 2 -> 1
                                                         3: 3 -> 2
```

Fig. 3. Tabular representation of **crossing**

the structure of the global system by means of the n-uple of numbers following the keyword `Components`. This n-uple indicates the templates of automata that constitute the system. In the case of **crossing**, the tuple `2 3` says that the system is composed by one instance of the automaton `2`, i.e. **barrier** and one instance of the automaton `3`, i.e. **signal**. The synchronizations among the components are specified as n+1-uple of numbers. Hence, in our case we have triples of numbers. A triple of the form $n0$: $n1$ $n2$ indicates that if the first component can perform action $n1$ and the second component can perform action $n2$ then the system can perform action $n0$. A 0 value for ni means that the corresponding component does not take part into the synchronization. The major simplification with respect to FC2 specifications is that both the transition definitions and the synchronization definitions use "plain" action indexes rather than generic expressions with action indexes as operands.

Both the FC2 and the tabular representations can be given as an input to the utility `tobsp` that returns a BSP program that represents the system.

BSP program of the specification

The BSP program that represents the **barrier** is given in Figure 4. Hereafter, we explain the main features of the language BSP. Examples refer to Figures 4, 5 and 7.

Comments are C-like, i.e. started with `/*` and ended with `*/`.

A *declaration* of the form `(def` *id* `(array` *n*`))` defines the identifier *id* to be a vector of *n* boolean variables. We will refer to an identifier declared in this way as an *array*. Thus arrays denote vectors of boolean variables. For example, `(def C1_a (array 2))`, `(def C1_px (array 1))` declare arrays ranging, respectively, on actions and present states of `C1_`. Boolean variables are represented with

```
(def C1_a (array  2))
(enum  2 0 (C1_a0 C1_a1 C1_a2 C1_a3))
(def C1_px (array  1))
(def C1_nx (array  1))
(enum  1 0 (C1_s1 C1_s2))
(def C1_S ( eqv C1_px C1_s1))
(def C1_R
  (defprocess
    (present_state C1_px)
    (next_state C1_nx)
    (transition
      (update C1_px)
      (eqv C1_px C1_s1)
      (eqv C1_a C1_a2)
      (eqv C1_nx C1_s1))
    (transition
      (update C1_px)
      (eqv C1_px C1_s1)
      (eqv C1_a C1_a1)
      (eqv C1_nx C1_s2))
    (transition
      (update C1_px)
      (eqv C1_px C1_s2)
      (eqv C1_a C1_a3)
      (eqv C1_nx C1_s1)) ))
```

Fig. 4. BSP program of **barrier**

BDD variables. Unless otherwise instructed BSP uses as BDD variable ordering the ordering in which boolean variables (arrays) are declared. It is possible to override this behaviour by instructing BSP to follow a user given BDD variable ordering.

A *declaration* of the form (**def** *id* (**record** $X_1 \ldots X_n$)) defines identifier *id* as the vector of boolean variables obtained by concatenating the vectors of boolean variables $X_1 \ldots X_n$. We will refer to an identifier declared in this way as a *record*. Note that no new BDD variable is created by such declaration. For example, (**def px (record C1_px C2_px)**) gives name **px** to the vector of boolean variables formed by the variables in **C1_px** or in **C2_px**.

A *declaration* of the form (**enum** *size offset* ($id_0 \ldots id_{k-1}$)) defines identifiers $id_0 \ldots id_{k-1}$ to be vectors of *size* boolean values. Identifier id_0 is the boolean representation of $offset \bmod 2^{size}$, id_1 is the boolean representation of $(offset + 1) \bmod 2^{size}$, etc. For example, (**enum 2 0 (C1_a0 C1_a1 C1_a2 C1_a3)**) assigns to **C1_a0**, **C1_a1**, **C1_a2**, **C1_a3** respectively the vectors [0 0], [1 0], [0 1], [1 1] (note: the leftmost bit is the least significant bit).

A *term* of *size* n is a vector of n boolean expressions. For example, **C1_a**, **C1_s1**, **px**, are terms of size, respectively, 2, 1, 3. More complex terms can be

built using boolean operators and quantifiers. For example, if `a` is a record or an array and `R` and `F_OF_1` are terms of size n then the following are terms of size n: `(and R F_OF_1)` (semantics: bitwise *and* of `R` and `F_OF_1`), `(exists a (and R F_OF_1))` (semantics: $(\exists$ `a` $(\texttt{R} \wedge \texttt{F_OF_1}))$). If t_1, t_2 are terms of size n then `(eqv` t_1 t_2`)` is a term of size 1 evaluating to 1 iff terms t_1, t_2 are bitwise equal.

A *definition* `(def` *id* t`)` assigns to *id* the value of term t. For example, `C1_S` denotes a boolean function which is 1 (true) iff `C1_px` is bitwise equal to `C1_s1`.

A *term* of the form `(defprocess ...)` is used to define the transition relation of a process (LTS). For example, `C1_R` is a boolean function which is 1 iff in the LTS `C1_` there is a transition labeled with `C1_a` from state `C1_px` to state `C1_nx`.

The transition relation for the parallel composition of two LTSs is obtained by defining in BSP the semantics of the parallel composition operator. For each automaton constituting the system (i.e. **barrier** and **signal**), a boolean function modeling its set of transitions is constructed. Similarly, the global system (i.e. **crossing**) transitions are modeled as a boolean function constructed starting from the transitions of the system components. Part of the BSP program that represents the **crossing** is given in Figure 5.

The properties

We have verified the following properties:

1. if a train leaves the level crossing (event `leaving_t`) then, first, it has to approach to the level crossing (event `approaching_t`);
2. if a train approaches to the level crossing then it must immediately leave the level crossing;
3. if a car approaches the crossing (event `approaching_c`) then no train can approach the crossing until a car leaves (event `leaving_c`) the crossing;
4. it is possible to have any number of cars approaching and leaving the crossing.

The μ-ACTL formulae that correspond to the previous properties are shown in Figure 6.

BSP programs of the properties

The translation from μ-ACTL to BSP is done by defining with BSP the semantics of μ-ACTL formulae. The translation of a single μ-ACTL formula into a list of boolean function occurs in two steps. First the μ-ACTL formula is translated into the μ-calculus. Then, the resulting μ-calculus formula is compositionally translated into a list of boolean functions (BSP program). The standard translation from μ-ACTL to μ-calculus can be found in [16]. Figure 7 shows the (simplified) BSP translation of the first formula in Figure 6.

```
/* definition of missing components */
(def a (array  3))
(enum  3 0 (a0 a1 a2 a3 a4))
(def px (record C1_px C2_px))
(def nx (record C1_nx C2_nx))
(def aa (record C1_a C2_a))
(def S (and C1_S C2_S))
(def R (or
  (exists aa (and
    (eqv a a1)
    C1_R
    (eqv C1_a C1_a1)
    C2_R
    (eqv C2_a C2_a1)))
  (exists aa (and
    (eqv a a2)
    C1_R
    (eqv C1_a C1_a2)
    C2_R
    (eqv C2_a C2_a2)))
  (exists aa (and
    (eqv a a3)
    (eqv C1_px C1_nx)
    C2_R
    (eqv C2_a C2_a3)))
  (exists aa (and
    (eqv a a4)
    C1_R
    (eqv C1_a C1_a3)
    (eqv C2_px C2_nx)))  ))
```

Fig. 5. Part of the BSP program of **crossing**

Results of model checking

Once that the BSP programs of the specification and of the logical formulae have been produced, for each formula that must be checked, i.e. for each boolean function `F_*_i`, a query of the form

```
(def check_fun_i (forall px (imp S F_*_i)))
(isone check_fun_i)
```

is added to the file containing the translations, and the BSP interpreter is called on the file. The BSP term `(isone check_fun_i)` asks BSP to check whether the boolean function `check_fun_i` is identically 1, in that case the formula is true. The results of the model checking of the formulae in Figure 6 are in Figure 8. Note that all the formulae are true.

```
~ (EF  <~ approaching_t> <leaving_t> true).

AG ( [approaching_t] AX {leaving_t} true).

AG ([approaching_c] A [true{~approaching_t} U  {leaving_c} true]).

nu FORM : <approaching_c | leaving_c> FORM.
```

Fig. 6. μ-ACTL formulae

```
(def F_3FFB_1 (exists nx (exists a (and R (eqv a a4)))))

(def F_2FF_1 (exists nx (and (exists a (and R (not (eqv a a1))))
                                        (compose F_3FFB_1 px nx))))

(def F_1F_1 (or F_2FF_1  F_0E_1))
(def F_0E_1 (exists nx (and (exists a (and R b1)) (compose F_1F_1 px nx))))

(def F_3F_1 (not F_0E_1))
```

Fig. 7. BSP translation of the first formula in Figure 6

4 Conclusions

We have presented a symbolic model checker, SAM, for an action-based temporal logics that directly performs verification over Labeled Transition Systems. To our knowledge, this is the first attempt in this direction, since previous symbolic model-checkers have been defined for state-based temporal logics such as CTL.

SAM is currently used in a number of formal validation projects, among which we recall, in particular, the validation of fault tolerance mechanisms defined for an architecture for dependable systems, developed inside the european project GUARDS [2]. Three fault-tolerant mechanisms (namely, Inter-Consistency algorithm, Fault-Treatment mechanism and Multi-Level Integrity mechanism), have been modeled using the tools of the JACK environment; the possible occurrence of faults has been modeled by introducing explicit "fault" actions in the model, and different fault assumptions have been modeled, in order to study the behaviour of the mechanism under different fault hypotheses. The satisfaction of some critical properties of the mechanism, both in presence of faults or not, is the object of the on-going validation effort. In a first phase of the project, we were able to verify the properties of the Inter-node consistency algorithm when designed for a three node GUARDS architecture. The model exhibited (after some abstraction) around 70000 states and was affordable by AMC, the traditional model checker for ACTL. When the algorithm for a four-node GUARDS architecture has been considered in the next phase of the project, it turned out that it was no more tractable by AMC, since the size was grown to 10^7 states. The fault treatment mechanism is the largest model we have generated inside

```
isone check_fun_1 AutVarOrd "check_fun_1" =
function check_fun_1 is identically 1

isone check_fun_2 AutVarOrd "check_fun_2" =
function check_fun_2 is identically 1

isone check_fun_3 AutVarOrd "check_fun_3" =
function check_fun_3 is identically 1

isone check_fun_4 AutVarOrd "check_fun_4" =
function check_fun_4 is identically 1
```

Fig. 8. Answers of the BSP interpreter

this project, amounting to $2 * 10^9$ states. Verification of critical properties is now in progress using SAM.

The integration of SAM within the JACK environment is under development, together with a friendly user interface.

Acknowledgments We thank Cinzia Bernardeschi and Antonella Santone for interesting discussions about the realization of this tool.

The development of SAM has seen at its beginning the contribution of Gioia Ristori, who has recently left us for a better world. She will however remain with us for ever.

References

1. D. Austry, G. Boudol. Algebre de processus et synchronization, *Theoretical Computer Science*, 1(30), 1984.
2. C. Bernardeschi, A. Fantechi, S. Gnesi. Formal Specification and Verification of the Inter-Channel Consistency Network, GUARDS Esprit project, Technical Report D3A4/6004c, 1997.
3. C. Bernardeschi, A. Fantechi, S. Gnesi, S. Larosa, G. Mongardi, D. Romano. A Formal Verification Environment for Railway Signaling System Design, in Formal Methods in System Design 12, 139-161, 1998.
4. A. Bouali, S. Gnesi, S. Larosa. The integration Project for the JACK Environment, Bulletin of the EATCS, n.54, pp.207-223, 1994. (see also http://rep1.iei.pi.cnr.it/projects/JACK.)
5. R.S. Boyer, J.S., Moore. "A Computational Logic", ACM Monograph Series, Academic Press, 1979.
6. R.E. Bryant. Graph Based algorithms for boolean function manipulation, *IEEE Transaction on Computers*, C-35(8), 1986.
7. J.R. Burch, E.M.Clarke, K.L. McMillan, D. Dill, J. Hwang. Symbolic Model Checking 10^{20} states and beyond, in Proceedings of LICS, 1990.

8. Clarke, E.M., Emerson, E.A., Sistla, A.P., "Automatic Verification of Finite–State Concurrent Systems Using Temporal Logic Specification," ACM Transaction on Programming Languages and Systems, 8(2):244-263, 1986.
9. R. Cleaveland, J. Parrow, B. Steffen. The Concurrency Workbench, in Proc. of Automatic Verification Methods for Finite State Systems, LNCS 407, pp. 24-37, 1990.
10. R. Cleaveland, S. Sims. The NCSU Concurrency Workbench, in Proc. of Computer Aided Verification, LNCS 1102, pp. 394-397, 1996.
11. R. De Nicola, A. Fantechi, S. Gnesi, G. Ristori. An action-based framework for verifying logical and behavioural properties of concurrent systems, *Computer Networks and ISDN Systems*, 25(7):761-778, 1993.
12. R. De Nicola, A. Fantechi, S. Gnesi, G. Ristori. Verifying Hardware Components within JACK, in Proceedings of CHARME '95, LNCS 987, pp. 246-260, 1995.
13. R. De Nicola, F. W. Vaandrager. Action versus State based Logics for Transition Systems, Proceedings Ecole de Printemps on Semantics of Concurrency, LNCS 469, pp. 407-419, 1990.
14. E.A. Emerson, J. Halpern. *"Sometime" and "Not never" revisited: On branching* versus linear time temporal logic, *JACM* 33:151-178, 1986.
15. E.A. Emerson, C. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus, in Proceedings of LICS, pp. 267-278, 1986.
16. A. Fantechi, S. Gnesi, G. Ristori. From ACTL to μ-Calculus, ERCIM Workshop on Theory and Practice in Verification, Pisa, December 9-11, 1992.
17. J.C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, M. Sighireanu. CADP: A Protocol Validation and Verification Toolbox, CAV'96, LNCS 1102, pp. 436-440, 1996.
18. G. Ferro. "AMC: ACTL Model Checker. Reference Manual", IEI-Internal Report B4-47, December 1994.
19. M. Hennessy, R. Milner. Algebraic Laws for Nondeterminism and Concurrency, *JACM* 32:137-161, 1985.
20. D. Kozen. Results on the Propositional μ-calculus, *Theoretical Computer Science*, 27:333-354, 1983.
21. E. Madelaine, R. de Simone. The FC2 Reference Manual, Available by ftp from `cma.cma.fr:pub/verif` as file *fc2refman.ps.gz*, 1993.
22. R. Milner. Communication and Concurrency, *Prentice-Hall International*, Englewood Cliffs, 1989.
23. G.D. Plotkin. A Structural Approach to Operational Semantics, Technical Report DAIMI FN-19, Aarhus University, Dep. of Computer Science, Denmark, 1981.
24. R. Pugliese, E. Tronci. Automatic Verification of a Hydroelectric Power Plant. FME'96, LNCS 1051, pp. 425-444, 1996.
25. V. Roy, R. De Simone. AUTO and Autograph, in Proceedings of the Workshop on Computer Aided Verification, LNCS 531, 65-75, 1990.
26. C. Stirling. An Introduction to modal and temporal logics for CCS, In Concurrency: Theory, Language, and Architecture, LNCS 391, 1989.
27. E. Tronci. Hardware Verification, Boolean Logic Programming, Boolean Functional Programming, in Proceedings of LICS, 1995.
28. E. Tronci. On Computing Optimal Controllers for Finite State Systems, Proc. of the 36th IEEE Conf. on Decision and Control, 1997.

Critical Systems Validation and Verification with CSP and FDR

Michael Goldsmith[1] and Irfan Zakiuddin[2]

[1] Formal Systems (Europe) Ltd, Keble Court, 26 Temple Street, Oxford OX4 1JS, UK
michael@fsel.com
[2] Defence Evaluation and Research Agency, St Andrews Road, Malvern, UK
i.zakiuddin@eris.dera.gov.uk

Abstract. Formal Systems and DERA have enjoyed a number of fruitful collaborations in recent years, especially in projects exploiting the FDR tool to analyse CSP models of systems. This paper presents an overview of the approach and some of the diverse applications to which it has been applied.

1 Introduction

Formal Systems has been involved in the application of formal techniques to a wide variety of problem domains throughout the 1990s, and this effort has included the development of tool support for analysis and verification. In recent years much of this work has centred around the use of CSP (Communicating Sequential Processes [5, 10]) to model systems, and subsequent analysis using the company's refinement checker FDR. The company was founded by academics from the Programming Research Group at Oxford University, together with applied mathematicians from US universities, to bring to commercial enjoyment the fruits of leading-edge research.

On the other hand, the System Assurance Group of DERA, the UK Defence Evaluation and Research Agency, runs a collection of research projects into techniques for analysing, verifying and certifying safety and security critical software-intensive systems. The research covers the full spectrum of software, digital electronic hardware and systems integration, and is aligned with real-world needs and applications throughout the system lifecycle, including fast prototyping and animation of requirements, formal analysis of critical software and hardware, system modelling to demonstrate safety and security integrity, and assembly of safety cases for certification and ongoing maintenance. A major part of the Group's activities is the provision of assessment and advice in support of Ministry of Defence and other projects, with particular focus on customer issues of acceptance and ownership rather than supplier problems of development.

The two organisations have enjoyed a successful relationship for several years, both in collaborative research and development projects, led and funded by DERA, and also with DERA as consumers of Formal Systems' tools offerings in support of their assessment role. Application domains have included the analysis

of cryptoprotocols (for key-exchange and authentication), where a number of new attacks have been discovered, and more recently fault-tolerant and safety-critical systems.

2 Technology

CSP is a powerful mathematical notation for describing and reasoning about systems built from a number of largely independent component processes which interact with each other by communication. Simultaneously, CSP provides a rich collection of mathematical models and reasoning methods which help to understand and use the notation.

The processes which are modelled typically present an abstract view of some concurrent computation, whether processes in operating-system terms, light-weight threads or nodes in a multi-processor configuration. It is also possible, however, to represent the behaviour of a human operator or a physical device connected in some way. This gives the potential for modelling the environment of a system, which provides a powerful mechanism both for restricting attention to anticipated (or anomalous) scenarios and for assisting in specification.

The theory gives a satisfactory treatment of the nondeterminism which naturally tends to arise in concurrent systems (and thus renders them intrinsically untestable), which can only be handled properly by an analytic approach. Any system exhibiting less nondeterminism, but otherwise compatible behaviour, can be substituted without affecting the satisfaction of existing specifications, as it is impossible to distinguish them from the less constrained implementation happening to resolve its internal choices that way. This relationship is called refinement, and combined with strong compositionality results allows CSP to be used in a step-wise and component-wise fashion, evading some of the problems which arise from the use of formalisms which require a global view of the system.

The notation was developed at Oxford University in the late 1970s and early '80s, and has seen extensive use in academic and practical applications since then. The advances in the theory, and in understanding how most effectively to apply it, which have been developed over the past decade have recently been captured in a major new text: Professor A.W. Roscoe's *Theory and Practice of Concurrency* [10]. This work takes full account of the potential for tools in support of the mathematics, and offers numerous examples which can be found, together with other material, at URL

`http://www.comlab.ox.ac.uk/oucl/publications/books/concurrency/`

By far the most significant development in this time (according to Roscoe) has been the emergence of tools to support both the teaching and industrial application of CSP. This has turned CSP from a notation used mainly for describing "toy" examples which could be understood and analysed by hand, into one which can and does support the description of industrial-sized problems and facilitate their automated analysis. In particular, the FDR model-checking

tool can, over a wide range of application areas, perform analyses and solve problems that are beyond most, if not all, humans.

There is a significant difference between the FDR approach to the analysis of CSP and that of most other model-checkers. That is, rather than comparing some abstract program describing a system against a specification written in some (frequently temporal) logic, FDR uses CSP for both roles. Its primary mode of operation is to compare a specification process with a purported refinement of it, to discover whether this is in fact the case. The compositionality of CSP (the transitivity of refinement and the monotonicity of its operators) thus allows complex results to be established step-wise. When a refinement is found not to hold, a counterexample is presented, and an interactive debugger allows the contributions of each (sub-)component to be explored. This is an essential feature for any tool with engineering aspirations!

This basic experiment can be used to address a range of questions. As well as simply comparing one system with another, simple coding tricks can capture a wide range of specifications as processes – refinement of such a specification process by a system is then equivalent to its satisfying the original specification. Alternatively, the tool can be used in a problem-solving mode, by specifying that the problem is insoluble and extracting a solution from the exhibited counter-example. Some properties which cannot be expressed as a refinement query also have direct support in the tool; in particular, whether a system is completely deterministic.

FDR is outwardly one of the explicit state exploration school of model-checkers, but the second-generation FDR2 also provides for the exploitation of a number of semantic-preserving state-space reduction operators, which can be applied hierarchically to system as it is composed. While the practical limit on the number of states that can be explored is of the rough order of 10^8, these compressions allow systems with much larger underlying state spaces to be analysed; the system discussed in Sect. 3.3 below has an unconstrained space of closer to 10^{50} states, and in suitably contrived examples systems with doubly exponential numbers of states can be handled in minutes.

CSP has been applied to a wide range of problems. Communication protocols are natural target, but it has also been used for analysing instruction flow interlocks in pipelined processors and human interactions with flight control systems. FDR grew out of a tool developed to meet verification demands for the Virtual Channel Processor of Inmos' T9000 transputer, and was swiftly applied to verifying the internal communications architecture of the Draper Laboratory TFTP fault-tolerant processor nodes. More recently, CSP backed by FDR has formed the core of the very successful DERA Strategic Research Programme on "Modelling and Analysis of Security Protocols" discussed in Sect. 3.1 and subsequent extensions of the work to address safety-critical issues, both at R&D and applied project levels (Sect. 3.2, Sect. 3.3). Another collaboration, involving Oxford University with the Smith Institute, London Underground and British Rail Research has led to ground-breaking advances in the modelling and analysis of Solid State Interlock signalling systems, proving the applicability of FDR-

based techniques on real geographical data previously believed intractable by any formal technique [12].

Continued experience has given better heuristic insights into the uses that can be made of the hierarchical compression functions in FDR, which rewrite transition systems to give denotationally equivalent state machines with, hopefully, many fewer states to explore. Combined with incremental algorithmic improvements (and increases in affordable hardware capability), these compressions bring hitherto unmanageable problems within the scope of formal analysis.

This ability to cope with larger systems is complemented by new research at Oxford University [6], into data independence and the exploitation of symmetry, which sanctions the analysis of relatively small systems to establish results of unbounded generality. Several of these techniques have long been used by practitioners, with only somewhat handwaving justification; but this work puts the intuitions on a sound formal basis, and adds support for more complex scenarios. Other academic work is addressing systems of arbitrary size by using FDR to establish base and step results in suitably framed inductions [3, 8].

The tools for CSP have now been extended to include an animator, ProBE, which complements FDR, requiring a lower level of user sophistication. ProBE is based on the same parser and operational semantics engine as the FDR compiler, and thus the tools accept common system description scripts (although many of the restrictions due to finiteness can be relaxed in ProBE). The animator provides a menu of initial actions for a given process term, and of resultant states for each action. A particular path from the root may be selected for further exploration, and projected onto the component processes. Further facilities include searches for the availability of a particular set of events, or for a stable state.

In essence, ProBE helps the user to gain comprehension of a CSP model through experimentation. In doing so, it may fulfil several roles: as a teaching aid, as part of the validation of a model of which FDR will establish important properties, or to illuminate the desired behaviour of a component to the engineer delegated to implement a system.

Further information on both tools can be found on the Formal Systems web site, at URL

`http://www.formal.demon.co.uk/`

The tools have a mixed user base of academic, industrial and governmental organisations, spread across four continents.

3 Applications

FDR differs from many formal methods tools in that it has, from the outset, been developed for and used in serious industrial application areas. The last few years have continued this trend, with significant new work by customers and collaborators as well as the developers.

Three key applications are discussed in more detail below. As well as its original use for architectural level verification of VLSI designs, other applications have included telecommunications systems, quality-of-service protocols and distributed simulation. Recently FDR has also been used to exhibit flaws in a number of distributed systems protocols, including one with a published "proof" of correctness and one from a mainstream textbook on the subject.

3.1 Security

Many of the recent advances in the technology were driven by work on information security, particularly the analysis of cryptoprotocols in a collaborative project led by DERA [7, 9, 4]. Techniques were developed for modelling a wide range of protocols, from Needham-Schroeder (both public-key and symmetric-key versions), through TMN and Wide-Mouth Frog to more algebraic examples such as Diffie-Hellman, Goss, and Li Gong's protocols based on one-way functions. Lowe's development of *CASPER* [2], a compiler from a custom cryptoprotocol description language into CSP scripts for FDR, has rendered such modelling quite straightforward: a matter of hours, rather than days, from starting to code to (partial) verification or compromise. There is active research in progress at a number of sites into various ways of extending the failure to find an attack in a limited size of system (as is required for finite-state analysis) to guarantees of freedom from such attacks in general.

Analysis has included considerations of timeliness and long-term denial of service, as well as immediate attacks on confidentiality and authentication. This work broke new ground in the field, and has been the inspiration of a considerable number of highly derivative efforts with other model-checkers.

Other work from the security domain, characterising noninterference in terms of determinism [11], has been adapted to apply to safety-critical analysis: in particular, the railway signalling work mentioned above [12].

3.2 Fault-Tolerance

The use of CSP backed by FDR for the analysis of fault-tolerant architectures and protocols extends back several years [1]. A recent example of interest to DERA involved the use of multiple non-volatile memories to preserve atomicity of updates in the event of power failure.

In this case it proved possible to make explicit the behaviour of a memory being written into at the time of failure in a number of ways: whether the whole might be corrupted, or only the addressed location; and whether corrupted data could be relied on to be recognised as such, as opposed to more Byzantine failures. The model produced allowed the effect of these parameters on the various claims made of the algorithm to be explored. In fact, despite the fact that the algorithm appeared to be based on the most favourable assumptions (global, detectable corruption), it turned out that the principal claim of atomicity was in fact achieved even in the most tricky case (local corruption to valid but incorrect data).

3.3 Safety-Critical Analysis

Fault-tolerance is only one aspect of safety-critical systems which has come under study in recent collaborations. Another example of interest to DERA consists of a number of communicating modules where, for safety, certain signals must only be generated after some other set (an authentication problem, in cryptoprotocol terms). The implementation has firing rules based only on the information available at each module, so many of the authentication requirements are mediated by several additional implementation signals.

In essence, this problem reduces to comparison of two inference systems: one with deductions among the "interesting" signals alone, and one involving all of them. It is reasonably straightforward to model these in the same way as the knowledge of the attacker in a cryptoprotocol scenario [4]. Abstraction of the implementation signals by concealment, followed by application of a powerful partial-order compression function, allows the desired relationships to be established within minutes.

So far, there is nothing which could not equally well (or perhaps better) be handled in a mechanised proof system or a logic programming language; but it would there be much harder to handle dynamic behaviour of the system. In particular, the CSP model is easily extended to analyse the resilience of the authentications to communication noise (so that one or more modules believe a signal has been sent before it has) or component failure (which in the worst case mean that all connected modules are misinformed) under a variety of error assumptions. More complicated queries can also be addressed, such as the effects of explicitly retracting an authorising signal.

This system presents a number of modelling challenges. For a start, its size (the implementation modelled naïvely would have some 10^{50} states) makes careful use (and justification, where necessary) of compressions essential, as discussed below. Equally, while it is reasonably straightforward to model the negative inferences arising from retraction, it is more difficult to determine what behaviour the specification should allow in this context.

Managing the Inference System. The basic structure of the system is, for reasons of compilation efficiency, a separate process representing each firing rule: these can be thought of as accumulating the necessary preconditions, and then signalling the conclusion when all have been seen. (The system actually implements disjunctive rules as well, but these are quite similar.) These processes synchronise on the signals which logically connect one to another, and those which do not appear in any given specification are abstracted by concealment, becoming autonomous, invisible, internal actions.

Although this structure of model has significant advantages, it does have a crucial practical drawback if implemented directly as described. Because of the way the CSP semantic models treat internal actions, in order to establish the refinement properties of such an abstracted system it is necessary to consider all possible combinations of reachable states. For example if two deductions may occur which do not impinge on one another, there are four configurations of

the inference system which need to be tested (where none, either one, or both have taken place), even though in our application the exact order of deductions should make no difference to the final outcome. This combinatorial explosion is clearly undesirable, and is made worse if visible signals become enabled while there remain hidden inferences to complete: each such event further increases the number of interleaved paths to explore.

As in the analysis of cryptoprotocols, however, we may make use of the specific properties of such inference systems. Since the deduction system is, in semantic fact, deterministic, despite the internal actions, we can use partial-order techniques to optimise the exploration: for this means that each state of the system has a unique final successor under internal actions, at least up to semantic equivalence. Our approach to simplifying the exploration is thus to consider not the raw parallel process described above, but the state machine which results from replacing any state by the semantically unique stable state reachable from it, and to eliminate the internal actions from our representation of the process altogether. In effect we evaluate the effect of hidden inferences before considering the system's visible interaction with the environment. This eager evaluation of transitions out of a single state does not, however, prevent our exploring the actual state space itself in a lazy fashion, so we need only explore sufficient of the system to establish or refute the claimed refinement property, and we can thus enjoy the best of both worlds.

The FDR 2 system provides a highly flexible interface for adding transformations on state machines, and the scheme for removing internal actions described here has been implemented using this facility. The resulting transformation is available as an external function *chase* in the FDR 2 input language.

References

1. N.A. Brock and D.M. Jackson. Formal Verification of a Fault Tolerant Computer. In *Proceedings of 1992 Digital Avionics Systems Conference*. IEEE, 1992.
2. `http://www.mcs.le.ac.uk/~glowe/security/casper/index.html`. World Wide Web page.
3. Sadie Creese. An inductive technique for modelling arbitrarily configured networks. MSc thesis, Oxford University Computing Laboratory, 1997.
4. Michael Goldsmith and Bill Roscoe. The perfect 'spy' for model-checking cryptoprotocols. In *Proceedings of DIMACS Workshop on Design and Formal Verification of Cryptographic Protocols*, Rutgers, 1997.
5. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.
6. Ranko S. Lazić. A Semantic Study of Data-independence with Applications to Mechanical Verification of Concurrent Systems. Technical report, Merton College, May 1997. A Dissertation Submitted for (and Winning!) the Oxford University Senior Mathematical Prize.
7. `http://www.comlab.ox.ac.uk/oucl/groups/security`. World-Wide Web page.
8. J.N. Reed, D.M. Jackson, B. Deianov, and G.M. Reed. Automated formal analysis of networks: FDR models of arbitrary topologies amd flow-control mechanisms. In *European Joint Workshop on Theory and Practice in Software; Fundamental Approaches to Software Engineering (ETAPS-FASE'98)*, Lisbon, March 1998.

9. A.W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *Symposium on Foundations of Secure Systems.* IEEE, 1995.
10. A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998. ISBN 0-13-6774409-5, pp. xv+565.
11. A.W. Roscoe, J.C.P. Woodcock, and L. Wulf. Non-interference through determinism. In *1994 European Symposium on Research in Computer Security*, number 875 in LNCS, pages 33–53. Springer, 1994.
12. Andrew Simpson. *Safety through Security.* DPhil, Oxford University Computing Laboratory, Trinity 1996.

UniForM Perspectives for Formal Methods

Bernd Krieg-Brückner

Bremen Institute of Safe Systems
TZI, FB3, University of Bremen
P.O. Box 330440, D-28334 Bremen
bkb@Informatik.Uni-Bremen.DE

Abstract. Trends for Formal Methods are reviewed and illustrated by several industrial applications: logical foundations of combination, verification, transformation, testing, and tool support. The UniForM Workbench is the background for highlighting experiences made over the past 20 years.

1 Introduction

This paper outlines some state of the art technology in Formal Methods and attempts to extrapolate towards the next century. The UniForM Workbench (Universal Formal Methods Workbench, cf. Krieg-Brückner et al. 1996) is developed by the Universities of Bremen and Oldenburg, and *Elpro*, Berlin, funded by the German Ministry for Education and Research, BMBF. In its present state, it provides a focal point to

- review experiences in the past decades developing languages, methods and tools,
- give an illustrative example of state of the art technology,
- evaluate preliminary experiences with industrial applications, and
- assess the industrial potential for the future and illuminate technological trends.

2 Logical Foundations

The first use of Formal Methods in software development, regarded by many as the most prominent, is for modelling, using a mathematically well-founded specification language. The need for modelling arises in many aspects and properties of software, or more generally systems: for the physical environment of a hybrid hardware / software system, for the timing behaviour and real-time constraints of an embedded system, for the hazards and safety requirements of a safety-critical system, for the concurrent interactions of a reactive system, for deadlock and livelock prevention, for performance and dependability analysis, for architectural and resource requirements, and, finally, at many stages of the software development process for requirements and design specifications, etc., to the implementation of a single module. The important distinction between property-oriented and model-oriented specifications is now universally accepted: the former are sufficiently loose to avoid over-specification, cap-

turing just the necessary properties and leaving freedom of choice for the implementor; the latter are geared towards describing a particular, operational implementation, giving sufficient detail to allow e.g. detailed performance analysis.

The second and equally important use of Formal Methods is to provide support for correct development, i.e. mathematical notions of refinement or implementation that guarantee correctness preservation w.r.t. the initial requirements, be it by the invent-and-verify paradigm, synthesis or transformation.

2.1 Combination of Formal Methods

How can we ever hope for a unique standard formalism to cover all the needs listed above? Instead, the solution is a variety of formalisms that complement each other, each adapted to the task at hand: specification languages and development methodologies, specific development methods or proof techniques, with a whole spectrum of tool support. Thus the challenge is to cater for correct combination of formalisms to

1. ensure correct transition from abstract to concrete specifications when switching between formalisms during the development process ("vertical composition"),
2. ensure correct combination of formalisms in a heterogeneous situation, e.g. combining concurrent and sequential fragments ("horizontal composition"),
3. attach various kinds of tools, e.g. for performance or deadlock analysis, and
4. provide "hooks" for specifications that do not affect semantics but allow the use of tools, e.g. ordering of operation symbols or equations for effective rewriting to prototype, or guidance for design choices w.r.t. expenditure of resources such as speed vs. space, or relative usage of operations of an abstract data type.

Such combinations are by no means easy to achieve. The need for research on the first two has been recognised and requires demanding mathematical foundations, such as advanced methods in category theory. This has lead to languages for "institution independent" heterogeneous composition of modules ("in the large", see e.g. Astesiano and Cerioli 1994, Tarlecki 1996, Diaconescu 1998); approaches for reasoning about correct composition of the logics capturing the semantics "in the small" (see e.g. Mossakowski 1996, Mossakowski et al. 1997, 1998b, Sernadas et al. 1998a, 1998b) introduce notions such as *embedding, translating* one formalism to another (cf. (1) above), *combination* of two formalisms, or *projecting* to either from the combination.

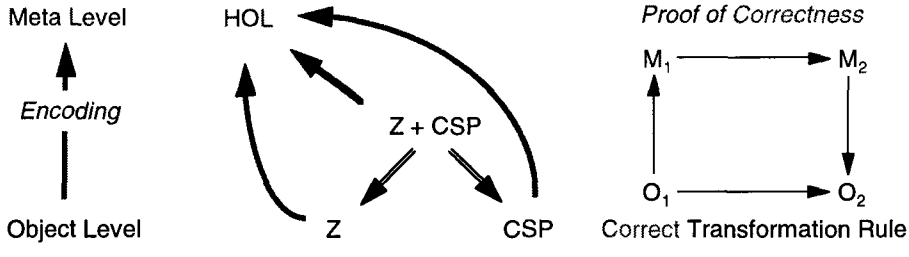


Fig. 1. Semantic Representation in UniForM

Semantic Representation. The approach of UniForM is to represent the semantics underlying a particular formalism or language in higher-order logic (HOL) as it is realised in the logical framework Isabelle (Paulson 1995). Fig. 1 shows a tiny Logic Graph for Z, CSP and their projections from the combination Z+CSP, plus the logic encoding into HOL at the meta level. Specifications in these languages are represented as theories in Isabelle and used for theorem proving with the verification system IsaWin on top of Isabelle (cf. section 3.1), and, as a basis for transformational development (cf. section 3.3), for proving the correctness of transformation rules.

HOL-Z, HOL-CSP and HOL-CASL. In HOL-Z, the logic of Z has been represented (cf. Kolyang et al. 1996a, 1996b, 1997, Kolyang 1997, Lüth et al. 1998b) and the mathematical tool kit has been proved correct (in co-operation with the ESPRESS project); this resulted in ca. 1k theorems, a 4k line proof script, and ca. 3 person-years of effort.

HOL-CSP represents the logic of CSP; a small but pervasive error in the 20 year old theory of CSP has been found and corrected (cf. Tej and Wolff 1997, Tej 1999). The process algebra has been proved correct; this resulted in ca. 3k theorems, a 17k line proof script, and ca. 3 person-years of effort. The example shows that such an endeavour is by no means trivial but pays off in the end. The proof of correctness of transformation rules, in particular, is now much easier.

The above statistics includes the effort of becoming familiar with the intricacies of Isabelle, and most of the effort went into the proof of the process algebra of CSP. A subsequent representation of the logics and static semantics of CASL basic specifications (including an intricate overloading resolution) only required about 1 person-year of effort (see Mossakowski et al. 1998a).

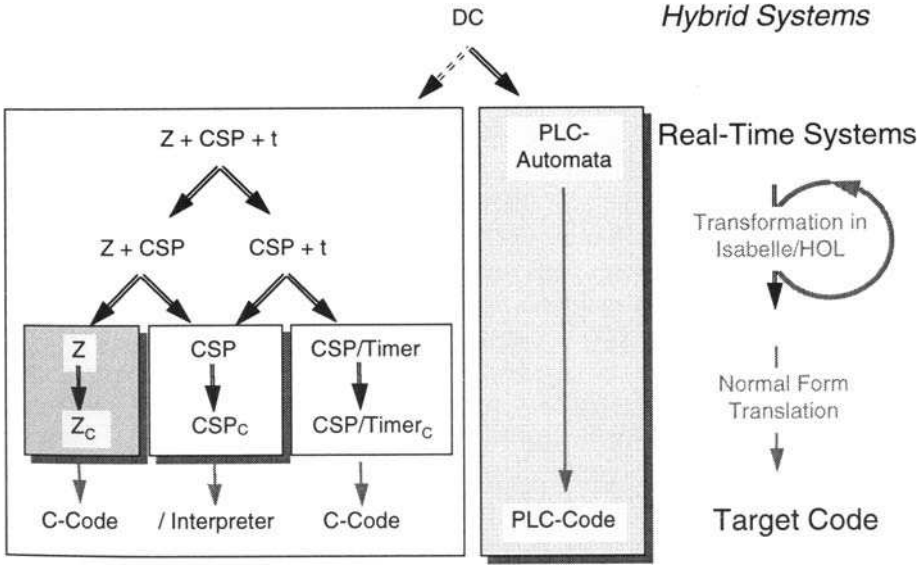


Fig. 2. Method Combination in UniForM

Reactive Real-Time Systems. The first instantiation of UniForM has been for Z and CSP since these are considered to be rather mature and relatively well established in industry. At the moment, we are working on methods ("structural transformations") to project not only from Z+CSP (actually Object-Z, cf. Fischer 1997), but also from CSP+t, i.e. CSP with real-time constraints, to CSP without such constraints on the one hand, and simple timer processes on the other, cf. fig. 2. Thus specialised methods can be used in the projected domains. This breakdown is also successfully used for testing of real-time and hybrid systems (cf. section 3.1).

A special class of hybrid systems, so-called PLC-Automata, generate real-time programs for Programmable Logic Controllers directly (cf. Dierks 1997, Tapken 1997, 1998). They have a semantics as DC-implementables, a subset of Duration Calculus (Zhou et al. 1992). Further extensions towards hybrid systems are planned.

2.2 Standard Family of Specification Languages

A standard formalism for all aspects of formal methods seems pragmatically undesirable (if not impossible) since a projection to a restricted and supposedly simpler formalism allows easier reasoning and specialised tools. However, standardisation should be aimed for in well-defined areas. IFIP WG 1.3 (Foundations of System Specification), based on more than 7 years of experience of the ESPRIT WG COMPASS, (cf. e.g. Krieg-Brückner 1996), started the Common Framework Initiative for Algebraic Specification and Development, CoFI.

CoFI, an international effort by primarily European groups, is developing a family of specification languages, a methodology guide and associated tools (see CoFI, Mosses 1997). The major language in this family, the Common Algebraic Specification Language CASL, has just been completed; it is the basis for sublanguages and extensions in the family. Its formal semantics only awaits a final revision. Various parsers exist as well as a prototype implementation of the static semantic analysis for basic specifications in Isabelle for the UniForM Workbench (Mossakowski et al. 1997); it allows theorem proving and will be the basis for transformational development.

CASL is a rather powerful and general specification language for first-order logic specifications with partial and total functions, predicates, subsorting, and generalized overloading (cf. CoFI, Cerioli et al. 1997). Sublanguages of CASL, in connection with the planned extensions towards higher-order, object-oriented and concurrent aspects, allow interfacing to specialised tools and mapping from/to other specification languages (cf. Mossakowski 1998); this aspect is crucial for its intended impact.

3 Verification and Validation

Formal Methods are meant for the development of dependable systems: apart from safety and security, aspects of availability, reliability, fault-tolerance, and a general adherence to functionality requirements are important. Thus correctness is only one aspect, but obviously at the heart of the matter. In particular in safety-critical do-

mains, application developers become increasingly aware of the importance of methods *guaranteeing* correctness w.r.t. a formal specification, the "contract".

3.1 Testing vs. Correctness Proofs

In many applications, complete formal development is regarded as unrealistic so far; the technology is only emerging. Often, only the critical kernel is formally developed But even with large systems, formal methods become increasingly important:

Verification, validation and testing. Peleska (1996) has developed a methodology and the tool kit VVT (cf. also Peleska and Siegel 1996, 1997), presently being integrated into the UniForM Workbench, that allows *automatic* testing. Test cases are generated from a real-time specification; they drive the completed hardware/software system as a "black box" in a hardware-in-the-loop configuration from a separate computer containing the test drivers, simulating a normal or faulty environment. The testing theory ensures, that each test will make an actual contribution, approximating and converging to a complete verification.

Even more important in practice is that thousands of lines of generated test cases can also be checked automatically against the formal specification; manual inspection is quite impossible. This approach is very cost-effective. It has been applied successfully to part of the case study of UniForM, a control computer on board of a train for railway control, developed by *Elpro*, Berlin (cf. Amthor and Dick 1997), and to an electric power control component of a satellite developed by OHB, Bremen.

Abstraction to verify special properties. Another team lead by Peleska (Buth et al. 1997, 1998, Urban et al. 1998) has developed a technique for abstracting from an existing program to verify the absence of deadlocks and livelocks. It was applied successfully to more than 100k lines of Occam implementing a safety layer of a fault tolerant computer to be used in the International Space Station Alpha developed by DASA RI, Bremen; thus it is scalable and applicable to realistic applications.

The concrete program is abstracted to a formal specification in CSP containing only the *essential communication behaviour*; the approach guarantees, that the proof for the abstract program implies the proved property for the concrete one. If the proof fails, the property does not hold, or the abstraction is not yet fine enough. The task is split into manageable subtasks by modularisation according to the process structure, and a set of generic composition theories developed for the application. The modules are then model-checked using the tool FDR (Formal Systems Ltd., Oxford).

The abstraction was done by hand; future research will focus on implementing formal abstraction transformations in the UniForM Workbench to support the process.

3.2 Model-Checking vs. Interactive Proofs

Model-checking is a very important technique in practice. The FDR tool is very useful for CSP, mostly for validating specifications, proving properties such as deadlock-freeness, and for development, proving the correctness of a refinement in the invent-

and-verify paradigm. But it can do more: the transition graph it generates can be interpreted at run-time; this technique has been used for the safety layer of a computer on-board a train (see above). The abstraction and modularisation method applied to the International Space Station, described in the preceding paragraphs, shows two things:

- Model-checking is extremely useful when the resp. data-types are essentially enumeration types and the systems small enough.
- For large systems, these properties are likely to be violated; reasoning about modularisation and composition properties is necessary; proof tools are desirable.

Thus both model-checking and (interactive) proofs should go hand in hand. In the UniForM Workbench, a link from the interactive proof tool (see next paragraph) to the FDR tool is presently being implemented.

Moreover, the experience of Haxthausen and Peleska (1998) when solving the train control problem in general has been, that reasoning about algebraic properties at a high level of abstraction is necessary, with subsequent refinements; model-oriented specifications and model-checking are not enough for this very practical problem that had defied a general solution thus far.

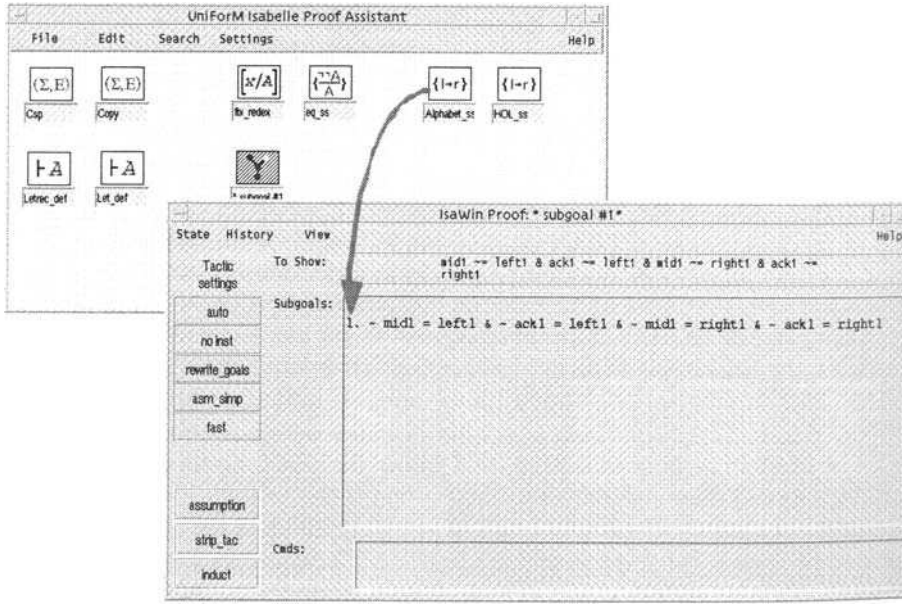


Fig. 3. The Isabelle Proof Assistant IsaWin in UniForM

A Window to Isabelle. The UniForM Workbench makes extensive use of the generic theorem prover Isabelle (Paulson 1995), and heavily relies on the possibilities for interaction and tactic definition. A graphical user interface, a "window to Isabelle", IsaWin, has been constructed that hides unnecessary details from the uninitiated user (see Kolyang et al. 1997, Lüth and Wolff 1998). Objects such as theories, substitutions, proof rules, simplification sets, theorems and proofs are typed (cf. fig. 3); icons

can be dragged onto each other or onto the manipulation window to achieve various effects. This graphical and gesture-oriented approach is as a major advance over the rather cryptic textual interface. In the example, a set of rewrite rules for simplification is dragged onto the ongoing proof goal in the manipulation.

Architecture of the UniForM Transformation and Verification System. In fact, theorem proving and transformation, both a form of deduction, are so analogous, that the UniForM Verification System IsaWin shares a substantial part of its implementation with the Transformation System TAS (cf. fig. 4, see Lüth and Wolff 1998, Lüth et al. 1998a). Like Isabelle, it is implemented in Standard ML; sml_tk (Lüth et al. 1996) is a typed interface in SML to Tcl/Tk; on top, the generic user interface GUI provides the appearance of fig. 3 and fig. 5. This basic system is then parametrized (as a functor in SML terminology) either by the facilities for theorem proving of IsaWin or those for transformation of TAS. In addition, both share focussing and manipulation of scripts, i.e. proofs or development histories.

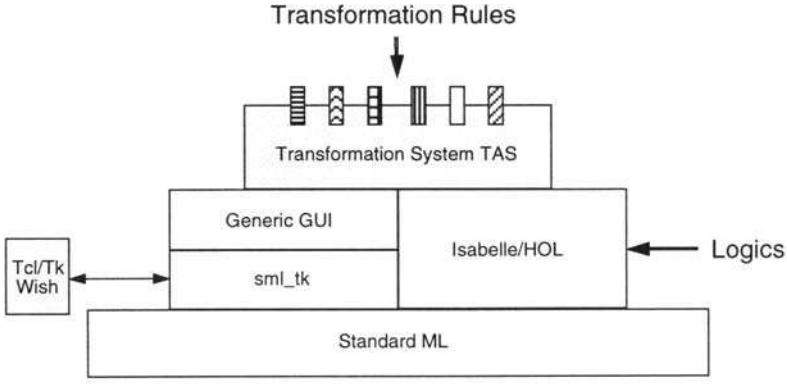


Fig. 4. Architecture of TAS, the UniForM Transformation System

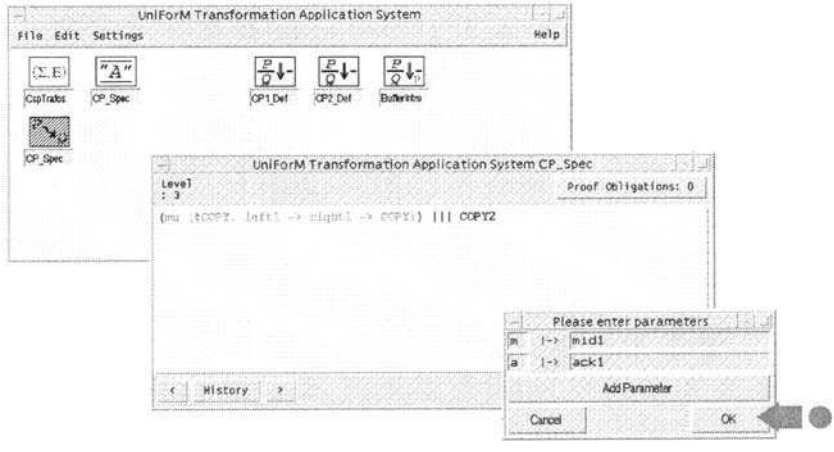


Fig. 5. Application of a Parametrized Transformation Rule

3.3 Transformation vs. Invent-and-Verify

Synthesis by Transformation. While the invent-and-verify paradigm is already supported by IsaWin, we definitely prefer synthesis-by-transformation over invent-and-verify as the pragmatically more powerful paradigm. First of all, the latter can be implemented by the former as a transformation rule that generates the necessary verification condition from the applicability condition. Secondly, this *automatic* generation of the required verification conditions is precisely one of the advantages of the transformational approach. The developer can concentrate on the development steps (viz. applications of transformation rules) first while the verification conditions are generated on the side and tabled for later treatment. Above all perhaps, single transformation rules and automated transformation methods embody development knowledge in a compact and accessible form like design patterns. Transformation rules preserve correctness; they can themselves be proved correct in UniForM against the semantics of the object language, e.g. at the level of the logic representation in HOL, cf. fig. 1.

TAS, the UniForM Transformation System. TAS may be parametrized by logics (i.e. semantic representation of the formalism involved) at the Isabelle level, and by transformation rules at the level of TAS itself, cf. fig. 4 (Lüth 1997). On top of the basic architecture that it shares with IsaWin, TAS provides icons for (program or specification) texts, transformation rules, possibly parametrized, and transformational developments in progress, in analogy to proofs (cf. shaded icon and manipulation window in fig. 5). In the example, a parametrized transformation rule is applied to the highlighted fragment denoted by focussing, and a window for the editing of parameters is opened. Once input of parameters is completed, the rule is applied, and a further proof obligation is possibly generated. A proof obligation may be discharged during or after the development by transferring it to IsaWin or another verification system such as a model checker (presently FDR).

The functionality of TAS subsumes that of a forerunner, the PROSPECTRA system (cf. Hoffmann and Krieg-Brückner 1993). However, the basis of Isabelle allows a more compact, more flexible and more powerful realisation: parametrisation by additional transformation rules is a matter of minutes (instantiation of a functor rather than recompilation of the whole system!); static semantic analysis can often be mapped to type checking of Isabelle; proof tactics can be defined as SML programs and often allow the automation of applicability conditions, such that much fewer residual verification conditions need to be interactively proved by the user.

Development History. Note also the History button that allows navigation in the development history, in particular partial undo for continuation in a different way. The whole development is documented automatically and can be inspected in a WWW browser, see fig. 6. The example shows the development of a communication protocol with send and receive buffers by a sequence of transformations in CSP.

Reusability of Developments. The development history is a formal object as well, (partial) replay is possible. A development can be turned into a new transformation rule by command; the generated verification conditions are then combined to a new

applicability condition. Combined with abstraction, *developments themselves* become reusable in new situations, not just their products, i.e. modules (cf. Lüth et al. 1999).

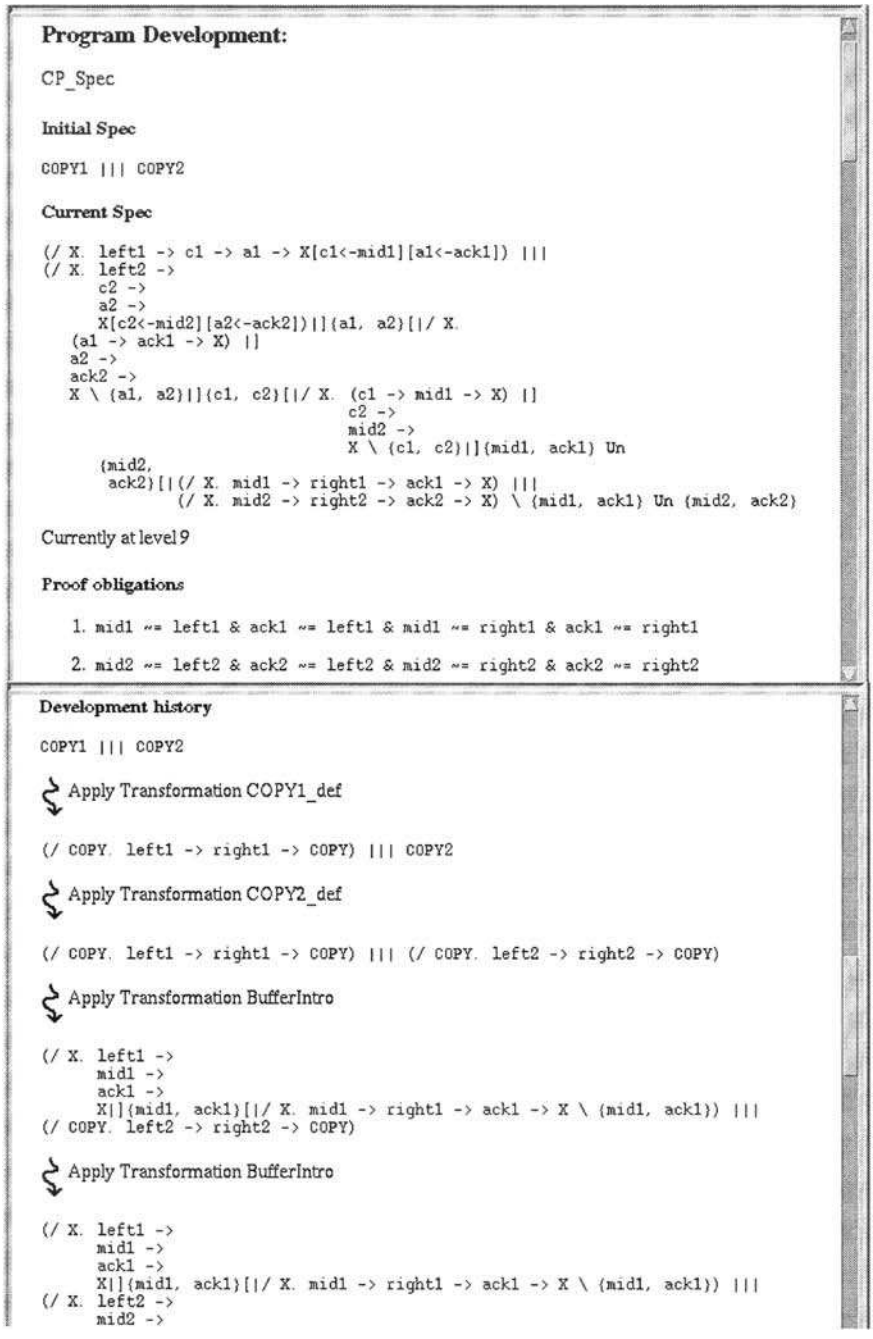


Fig. 6. Initial and Current Specification, Proof Obligations, and Development History

4 Integration into the Software Life Cycle

Integration of Formal Methods into Existing Process Models is important for success in industry. The Software Life Cycle Process Model V-Model (1997), originally a German development standard, has become internationally recognised. As many such standards, it loads a heavy burden on the developer by prescribing a multitude of documents to be produced. Thus tool support is essential to

1. tailor the V-model first to the needs of a particular enterprise, then
2. tailor the V-model to the special project at hand, fixing methods and tools,
3. support its enactment guiding and controlling the use of methods and tools, and
4. provide automatically generated development documents.

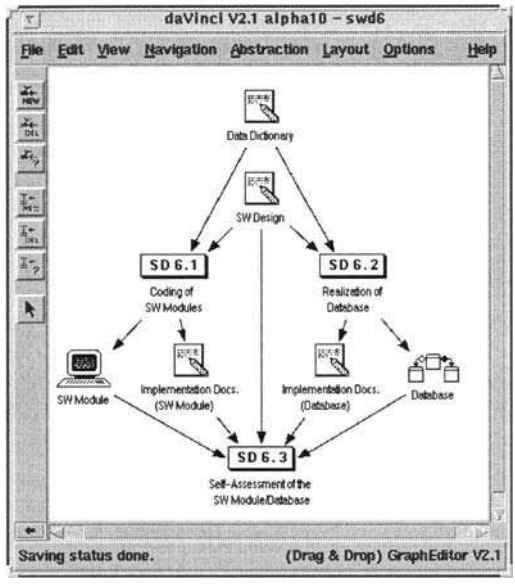


Fig. 7. Example of a V-Model Process Graph as supported by the UniForM Workbench

Blank Purper and Westmeier (1998) are presently developing the Graphical Development Process Assistant, adapting the V-model to formal methods, where development and quality assurance are intimately related. The V-model is presented as a heavily interwoven hypertext document, generated from a common database, and tools support items 1 to 4 above; cf. also fig. 7. Integration into a development environment such as the UniForM Workbench allows the coordination with its methods and tools (item 3). Tools themselves, on the other hand, can generate development documents in conformance with the V-model (cf. item 4), such as the development history of fig. 6.

Isolation of Dependable Parts is an issue that is at the heart of the combination problem; at the same time it offers possibilities for a kind of "divide and conquer" ap-

proach since methods and tools can be tailored to more specialised problems, cf. the projection into fragments with and without time for real-time systems in section 2.1.

Combination of Conventional, Semi-Formal and Formal Techniques arises naturally when interfacing with other methods in the context of the V-model. Safety considerations, and thus the employment of formal methods, will often be restricted to parts of a system. Ideally, graphical interfaces will give the illusion of working with an informal method while an underlying formal semantics provides hooks to the use of formal methods (cf. the PLC-Automata in section 2.1); this area has great potential.

At the same time, it is sometimes advisable to flip back and forth between informal techniques at a high level of abstraction, e.g. requirements analysis, and formal methods, once more detail is required; complete formalisation might be premature and rather a burden, but formal methods are already useful at an early stage to support the analysis. An example is the specialisation of fault trees for hazard analysis to develop safety requirements and safety mechanisms, cf. (Lankenau et al. 1998).

5 Tool Support

Large, Integrated Systems become unmaintainable dinosaurs very fast, in particular when several distant development teams are trying to coordinate, and maintenance threatens to cease.

Collections of Loosely Coupled Specialists are the obvious solution – unfortunately often *large* specialised tools, given the complexity of formal methods. It is most important that each tool can be developed and maintained by one or very few persons such that an overview over its functionality and integrity can be obtained and maintained in one head, not necessarily the same over time.

Increase of Productivity by Functional Languages. It is quite obvious that we should use formal methods eventually to produce our own tools; but is this realistic for really large systems? and during the initial bootstrapping phase of developing the first tools that allow scaling up? The answer is to use programming languages whose

- high-level aids self-documentation and maintenance,
- features for genericity instigate compactness and reuse,
- modularisation, information-hiding and inheritance features encourage structuring,
- static properties such as strong typing allow comprehensive checks,
- implementation permits separate compilation and, ideally, interpretation.

Our experience has been best with functional programming languages so far; we estimate the increase of productivity over, say, C, to a factor of 3. Without them, the development of large, non-trivial tools over a period of several years would have been impossible in an academic environment. TAS and IsaWin are extensions of Isabelle and comprise about 25k lines of SML; the graph visualisation system *daVinci* with thousands of users was developed by Fröhlich and Werner (1994, see also (Fröhlich 1997), and cf. fig. 7) over a period of 5 years comprising about 35k lines of a func-

tional language developed at Bremen, plus about 10k lines of C for interfacing; the tool integration framework of the UniForM Workbench, developed (see below), contains about 50k lines of Haskell.

Integration of Tools in the UniForM Workbench is described in detail in a companion paper (Karlsen 1998a; see also 1998b). Control and data integration is provided by the Subsystem Interaction Manager; based on the UniForM Concurrency Toolkit, tools interact like communicating concurrent agents and are, in general, loosely coupled by intermittent adaptors (cf. Karlsen 1997a, 1997b). The Repository Manager is a graphical interface (using *daVinci*) to a public domain version of the industry standard Portable Common Tool Environment and provides version and configuration control, etc. (cf. (PCTE 1994), (H-PCTE 1996), and (Karlsen and Westmeier 1997)).

The User Interaction Manager provides presentation integration, incorporating interfaces to *daVinci* and its extension Forest, a WWW-browser, and Tcl/Tk for window management. In particular the latter two become much more manageable and homogeneous by encapsulation into a typed, high-level interface in Haskell.

Haskell is the internal integration language; thus even higher-order objects and processes can be transmitted as objects. External tools are wrapped into a Haskell interface; however, we plan an adaptation of the Interface Definition Language of the industry standard CORBA to Haskell that will then shortly open more possibilities to integrate tools in, say, C, C++, or Java.

The Most Promising Architectures for Development Tools are those that avoid self-containment and allow integration with others. The possibility for control and data integration of a tool as an "abstract data type" is the most important (and not obvious since the tool may e.g. not allow remote control and insist on call-backs); integration of persistent data storage in a common repository is next (this may require export and import w.r.t. local storage); presentation integration with the same user interface is last - in fact it is most likely that the tool has its own graphical user interface. However, interactive Posix tools usually have a line-oriented interface that can easily be adapted (Karlsen 1997b).

This way, a graphical interface to HUGS was developed in a matter of weeks. Isabelle, IsaWin and TAS have been integrated, and a Z-Workbench with various tools has been instantiated from the UniForM Workbench (Lüth et al. 1998a, 1998b).

6 Conclusion

Formal methods are on the verge of becoming competitive in an industrial context, they are even cost-effective now when we consider their use in automated testing or deadlock detection for software that has been developed in a conventional way.
We have presented the following theses:

- Combination of various methods is essential; more basic research is needed here.
- Treatment of real-time and hybrid systems, in particular, is just emerging.
- Testing benefits greatly from the use of Formal Methods.

- Model checking and interactive verification of properties both are necessary.
- Synthesis by transformation has great potential over invent-and-verify.
- Reuse of formal developments is much more powerful than reuse of products.
- Tools for process models should support tailoring and generation of documents.
- The success of formal methods depends on the usability of support tools.
- Development environments should integrate tools in a loosely coupled way.

7 References

Amthor, P., Dick, S. (1997): Test eines Bordcomputers für ein dezentrales Zugsteuerungssystem unter Verwendung des Werkzeuges VVT-RT. 7. *Kolloquium Software-Entwicklung Methoden, Werkzeuge, Erfahrungen: Mächtigkeit der Software und ihre Beherrschung,* Technische Akademie Esslingen.

Astesiano, E. and Cerioli, M. (1994): Multiparadigm Specification Languages: a First Attempt at Foundations, In: C.M.D.J. Andrews and J.F. Groote (eds.), Semantics of Specification Languages (SoSl'93), Workshops in Computing, pp. 168-185. Springer.

Blank Purper, C., Westmeier, S. (1998): A Graphical Development Process Assistant for Formal Methods. In: *Proc. VISUAL'98 (short papers),* at ETAPS'98, Lisbon. `http://www.tzi.de/~uniform/gdpa`

Buth, B., Kouvaras, M., Peleska, J., Shi, H. (1997): Deadlock Analysis for a Fault-Tolerant System. In Johnson, M. (ed.): Algebraic Methodology and Software Technology. *AMAST'97.* LNCS 1349. Springer, pp. 60-75.

Buth, B., Peleska, J., Shi, H. (1998): Combining Methods for the Livelock Analysis of a Fault-Tolerant System. (submitted to AMAST'98)

Cerioli, M., Haxthausen, A., Krieg-Brückner, Mossakowski, T. (1997): Permissive Order-sorted Partial Logic in CASL. *AMAST'97.* LNCS 1349. Springer.

CoFI: The Common Framework Initiative for Algebraic Specification. `http://www.brics.dk/Projects/CoFI`

Diaconescu, R. (1998): Extra Theory Morphisms for Institutions: logical semantics for multi-paradigm languages. *J. Applied Categorical Structures* (to appear).

Dierks, H. (1997): PLC-Automata: A New Class of Implementable Real-Time Automata. Proc. ARTS'97. *LNCS 1231,* pp. 111-125. Springer.

Fischer, C. (1997): CSP-OZ: A Combination of Object-Z and CSP. In: *Proc. Formal Methods for Open Object-Based Distributed Systems (FMOODS '97).*

Fröhlich, M. (1997): Inkrementelles Graphlayout im Visualisierungssystem *daVinci*. Dissertation. *Monographs of the Bremen Institute of Safe Systems 6,* Aachen, Shaker.

Fröhlich, M., Werner, M. (1994): The interactive Graph-Visualization System *daVinci* – A User Interface for Applications. Informatik Bericht Nr. 5/94, Universität Bremen. updated documentation: `http://www.tzi.de/~daVinci`

H-PCTE (1996): The H-PCTE Crew: H-PCTE vs. PCTE, Version 2.8, Universität Siegen.

Haxthausen, A. E., Peleska, J. (1998): Formal Development and Verification of a Distributed Railway Control System. In *Proc. 1st FMERail Workshop,* Utrecht, The Netherlands, (to appear).

Hoffmann, B., Krieg-Brückner, B. (eds.) (1993): *PROgram Development by Specification and Transformation, The PROSPECTRA Methodology, Language Family, and System.* LNCS 680. Springer. `http://www.tzi.de/~prospectra`

Karlsen, E. W. (1997a): The UniForM Concurrency ToolKit and its Extensions to Concurrent Haskell. In: O'Donnåld, J. (ed.): *GFPW'97, Glasgow Workshop on Functional Programming '97,* Ullapool.

Karlsen, E. W. (1997b): Integrating Interactive Tools using Concurrent Haskell and Synchronous Events, In *ClaPF'97, 2nd Latin-American Conference on Functional Programming,* La Plata, Argentina.

Karlsen, E. W. (1998a): The UniForM WorkBench - a Higher Order Tool Integration Framework. In: *International Workshop on Current Trends in Applied Formal Methods*. LNCS. Springer (*this volume*).

Karlsen, E. W. (1998b): Tool Integration in a Functional Setting. Dissertation. Universität Bremen. 364pp. *(to appear)*.

Karlsen, E. W., Westmeier, S. (1997): Using Concurrent Haskell to Develop User Interfaces over an Active Repository. In *IFL'97, Implementation of Functional Languages 97*, St. Andrew, Scotland, September 10-12, 1997. LNCS 1467. Springer.

Kolyang (1997): HOL-Z, An Integrated Formal Support Environment for Z in Isabelle/HOL. Dissertation. *Monographs of the Bremen Institute of Safe Systems 5,* Aachen, Shaker.

Kolyang, Lüth, C., Meyer, T., Wolff, B. (1997): TAS and IsaWin: Generic Interfaces for Transformational Program Development and Theorem Proving. In Bidoit, M. and Dauchet, M. (eds.): *Theory and Practice of Software Development '97.* LNCS 1214. pp. 855-859. Springer.

Kolyang, Santen, T., Wolff, B. (1996a): A Structure Preserving Encoding of Z in Isabelle/HOL. In *Proc. Int'l Conf. on Theorem Proving in Higher Order Logic* (Turku). LNCS 1125. Springer. `http://www.tzi.de/~kol/HOL-Z`

Kolyang, Santen, T., Wolff, B. (1996b): Correct and User-Friendly Implementations of Transformation Systems. In: Gaudel, M.-C., Woodcock, J. (eds.): *FME'96: Industrial Benefit and Advances in Formal Methods*. LNCS 1051, pp. 629-648. Springer.

Krieg-Brückner, B. (1996): Seven Years of COMPASS. In: Haveraaen, M., Owe, O., Dahl, O.-J. (eds.): *Recent Trends in Data Type Specification.* Proc. 11th ADT/COMPASS Workshop (Oslo 1995). LNCS 1130, pp. 1-13. Springer. `http://www.tzi.de/~compass`

Krieg-Brückner, B., Peleska, J., Olderog, E.-R., Balzer, D., Baer, A. (1996): UniForM, Universal Formal Methods Workbench. in: Grote, U., Wolf, G. (eds.): *Statusseminar des BMBF: Softwaretechnologie.* Deutsche Forschungsanstalt für Luft- und Raumfahrt, Berlin 337-356. See also `http://www.tzi.de/~uniform`

Lankenau, A., Meyer, O., Krieg-Brückner, B. (1998): Safety in Robotics: The Bremen Autonomous Wheelchair. In: *Proc. AMC'98, 5th Int. Workshop on Advanced Motion Control*, Coimbra, Portugal 1998. ISBN 0-7803-4484-7, pp. 524-529.

Lüth, C. (1997): Transformational Program Development in the UniForM Workbench. *Selected Papers from the 8th Nordic Workshop on Programming Theory*, Oslo, Dec. 1996. Oslo University Technical Report 248.

Lüth, C. and Wolff, B. (1998): Functional Design and Implementation of Graphical User Interfaces for Theorem Provers. (to appear in *Journal of Functional Programming*).

Lüth, C., Karlsen, E. W., Kolyang, Westmeier, S., Wolff, B. (1998a): Tool Integration in the UniForM WorkBench. In Berghammer, B., Buth, B., Berghammer, R., Peleska, J. (eds.): Tools for System Development and Verification. Workshop, Bremen, July 1996. *Monographs of the Bremen Institute of Safe Systems 1,* Aachen, Shaker.

Lüth, C., Karlsen, E. W., Kolyang, Westmeier, S., Wolff, B. (1998b): HOL-Z in the UniForM WorkBench - a Case Study in Tool Integration for Z. *Proc. ZUM'98, 11th International Conference of Z Users,* LNCS 1493, pp. 116-134. Springer.

Lüth, C., Shi, H., Krieg-Brückner, B. (1999): Abstraction in Transformational Developments. (submitted for publication).

Lüth, C., Westmeier, S., Wolff, B. (1996): sml_tk: Functional Programming for Graphical User Interfaces. Informatik Bericht Nr. 8/96, Universität Bremen. `http://www.tzi.de/~cxl/sml_tk`

Mossakowski, T. (1996): Representations, Hierarchies and Graphs of Institutions. Dissertation. *Monographs of the Bremen Institute of Safe Systems 2,* Aachen, Shaker.

Mossakowski, T. (1998): Translating other specification languages to CASL. *Recent Trends in Algebraic Development Techniques*. WADT'98, Lisbon. LNCS. (to appear)

Mossakowski, T., Kolyang, Krieg-Brückner, B. (1998a): Static Semantic Analysis and Theorem Proving for CASL. In Parisi-Pressice, F. (ed.): *Recent Trends in Algebraic Development Techniques*. WADT'97, LNCS 1376, pp. 333-348. Springer.

Mossakowski, T., Tarlecki, A., Pawlowski, W. (1997): Combining and Representing Logical Systems, In . Moggi, E. and Rosolini, G. (eds.): *Category Theory and Computer Science,* 7th Int. Conf. LNCS 1290, pp. 177-196. Springer.

Mossakowski, T., Tarlecki, A., Pawlowski, W. (1998b): Combining and Representing Logical Systems Using Model-Theoretic Parchments. In Parisi-Pressice, F. (ed.): *Recent Trends in Algebraic Development Techniques.* WADT'97, LNCS 1376, pp. 349-364. Springer.

Mosses, P. (1997): CoFI: The Common Framework Initiative for Algebraic Specification and Development. In Bidoit, M. and Dauchet, M. (eds.): *Theory and Practice of Software Development '97.* LNCS 1214, pp 115-137. Springer.

Paulson, L. (1995): *Isabelle: A generic theorem prover.* LNCS 828. Springer.

PCTE (1994): European Computer Manufacturers Association: *Portable Common Tool Environment (PCTE), Abstract Specification,* 3rd edition, ECMA-149. Geneva.

Peleska, J. (1996): Formal Methods and the Development of Dependable Systems. Bericht 1/96, Universität Bremen, Fachbereich Mathematik und Informatik, 72p. `http://www.tzi.de/~jp/papers/depend.ps.gz`

Peleska, J., Siegel, M. (1996): From Testing Theory to Test Driver Implementation. in: M.-C. Gaudel, J. Woodcock (eds.): *FME'96: Industrial Benefit and Advances in Formal Methods.* LNCS 1051. Springer, pp. 538-556.

Peleska, J., Siegel, M. (1997): Test Automation of Safety-Critical Reactive Systems. *South African Computer Jounal 19,* pp. 53-77.

Sernadas, A., Sernadas, C., Caleiro, C. (1998a): Fibring of logics as a categorial construction. *Journal of Logic and Computation8 (10),* pp. 1-31.

Sernadas, A., Sernadas, C., Caleiro, C., Mossakowski, T. (1998b): Categorical Fibring of Logics with Terms and Binding Operators. In Frontiers of Combining Systems - FroCoS'98. Kluwer Academic Publishers. To appear in Applied Logic Series.

Tapken, J. (1997): Interactive and Compilative Simulation of PLC-Automata. In: Hahn, W. and Lehmann, A. (eds.): *Simulation in Industry,* ESS'97. SCS, pp. 552-556.

Tapken, J. (1998): MOBY/PLC – A Design Tool for Hierarchical Real-Time Automata. System Demo. In: Astesiano, E. (ed.): *Proc. First Int'l Conf. on Fundamental Approaches to Software Engineering, FASE'98,* at ETAPS'98, Lisbon. LNCS. pp326-330. Springer.

Tarlecki, A. (1996): Moving between logical systems. In M. Haveraaen and O. Owe and O.-J. Dahl (eds.): *Recent Trends in Data Type Specifications.* 11th Workshop on Specification of Abstract Data Types. *LNCS 1130,* pp. 478-502. Springer.

Tej, Haykal (1999): HOL-CSP: Mechanised Formal Development of Concurrent Processes. Dissertation. *Monographs of the Bremen Institute of Safe Systems.* (forthcoming)

Tej, H. and Wolff, B. (1997): A Corrected Failure-Divergence Model for CSP in Isabelle / HOL. *Formal Methods Europe, Proc. FME'97,* Graz. LNCS 1313, pp. 318-337. Springer.

Urban, G., Kolinowitz, H.-J., Peleska, J. (1998): A Survivable Avionics System for Space Applications. in *Proc. FTCS-28, 28th Annual Symposium on Fault-Tolerant Computing,* Munich, Germany, 1998.

V-Model (1997): *Development Standard for IT Systems of the Federal Republic of Germany.* General Directives 250: Process Lifecycle; 251: Methods Allocation.

Zhou, C., Hoare, C.A.R., Ravn, A.P. (1992): A Calculus of Durations. *Information Processing Letters 40(5)* pp. 269-276.

The UniForM WorkBench
A Higher Order Tool Integration Framework

Einar W. Karlsen*

Bremen Institute of Safe Systems (BISS)
TZI, FB3, Mathematik und Informatik
Universität Bremen, Germany
ewk@informatik.uni-bremen.de

Abstract. The UniForM Workbench is an open ended tool integration framework for developing (formal) Software Development Environments (SDE) from the basis of pre-fabricated off-the-shelf development tools. The integration framework provides support for data, control and presentation integration as well as utilities for wrapping Haskell interfaces around existing development tools. Entire SDE's are then glued together on the basis of these encapsulations using Concurrent Haskell as the integration language, thus allowing integration to be done at a level of abstraction that is very close to the one offered by constructive formal specifications. So far, the integration framework has successfully been used to integrate tools for Haskell program development as well as specification and proof tools for Z specifications.

During the 80's there were several attempts to provide environments for synthesizing *tightly integrated* SDE's from the basis of abstract language specifications. The Synthesizer Generator (CSG) [46], Gandalf [13], PSG [2] and IPSEN [28] are prominent examples of this approach. They are not perfect solutions, due either to the development costs involved in requiring tools to be developed from scratch in a homogeneous language framework or due to the inapplicability of the language framework itself (e.g. the attribute grammars of CSG) in handling the problem domain proper (e.g. proof tools) [25]. More recent systems, such as Centaur [5] and the ASF+DSF environment [8], are more open by providing a toolbus for hooking in foreign tools. However, both approaches lack the required features for supporting control and data integration "in the large" and "in the many".

In the context of a formal program development environment, no single tool or language is believed to be sufficiently powerful enough to support an entire formal development process, rather it is more likely that several loosely coupled and pre-fabricated tools are put together, each supporting a single dedicated task of the overall development [26]. This may suggest that SDE's for formal methods should be based on a *loosely coupled* architecture [27,48] where pre-fabricated

* This work has been supported by the German Ministry for Education and Research (BMBF) as part of the project UniForM under grant No. FKZ 01 IS 521 B2.

tools from different sources are integrated in a *tool integration framework* such as the one outlined by the ECMA Standard [9]. Unfortunately, until now, these integration frameworks are extremely expensive commercial systems outruling any use in academia.

The UniForM Workbench [26] provides a public domain solution to the problem by offering a tool integration framework in support of data, control and presentation. It has been designed according to international standards and employs existing public domain tools to support the integration process. The lazy functional language Haskell [15], extended with a model to concurrency inspired by process algebras, takes the role of the central integration language offering features very close to the ones employed when formulating constructive formal specifications.

The remainder of this paper is organized as follows. The architecture of the UniForM Workbench is presented in the next section. An instantiation in the form of a workbench for specification and proof in Z [49] is presented in section 2. The integration language Haskell, and its relevance in serving as a tool integration language, is discussed in section 3. The various components of the tool integration framework supporting control, data and presentation integration are discussed in the sections 4 to 7. The last section discusses the results and the ideas for future work.

1 System Architecture

The UniForM Workbench has been designed according to the guidelines of the ECMA Reference Model [9], which outlines the abstract functionality required to support the various dimensions of a tool integration process, i.e. data, control and presentation integration. The architectural design of the UniForM Workbench, as an instantiation of the ECMA architecture for formal methods within the UniForM project, is documented in [26, 24] (see Figure 1). *Data integration* addresses the issue of sharing data between tools and the persistent storage of software objects such as specifications, proofs and programs. It is mainly provided by the *Repository Manager*. *Control integration* is concerned with communication and inter-operation among tools and between tools and the integration framework. It is provided by the *Subsystem Interaction Manager*. *Presentation integration* addresses the issue of tool appearance and user interaction, i.e. look & feel. It is provided by the *User Interaction Manager*.

The UniForM Workbench has been designed according to a number of basic principles and ideas [19]. Entire SDE's are constructed in the integration framework by *encapsulating* existing development tools such as editors, model checkers and proof tools. The workbench uses *Concurrent Haskell* [15, 40] as an *integration language*, i.e. as *glueware* in integrating the tools. Haskell interfaces are wrapped around existing tools using the *Integration Manager* first. The missing bits and pieces required to provide full integration with the rest of an SDE, are then expressed at a very high level of abstraction using Haskell. During the encapsulation process, the tool is set up to work with persistent objects of the

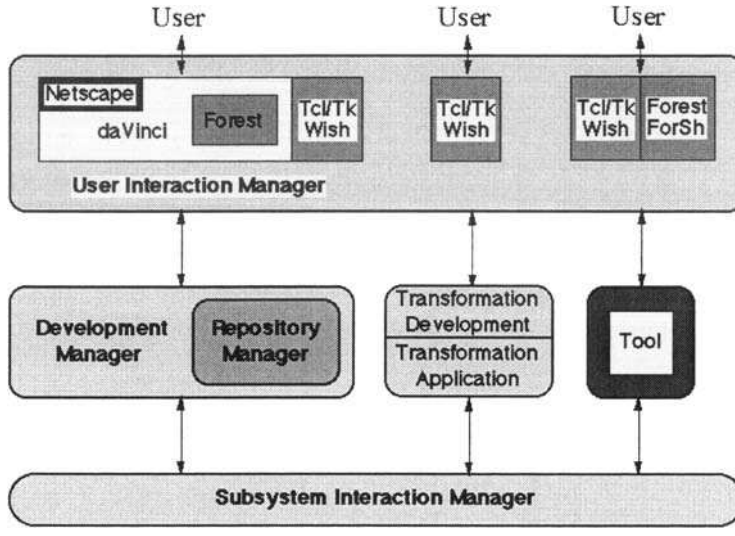


Fig. 1. UniForM Workbench Architecture

Repository Manager, rather than plain files of the file system. Control integration with other tools is made feasible by using the Subsystem Interaction Manager. User interfaces may in turn be provided for non-graphic tools and for persistent software objects using the User Interaction Manager.

An integrated SDE is viewed as a reactive, event driven system that reacts to user interactions, repository change notifications, individual tool events or operating system events. The Subsystem Interaction Manager plays the role of the central control component in this architecture. The approach to event handling is very close to the model of process algebras, in order to narrow the gap between formal specifications on the one hand and running programs on the other.

The workbench adheres to *existing standards* such as the ECMA Reference Model, the Portable Common Tool Environment (PCTE) [10], CORBA [37] (or COM [37, 6]), and Motif [11]. It has been constructed from *off-the-shelf public domain* implementations of these standards, such as H-PCTE [7], Tk [38] and daVinci [12], among others. These components have then been integrated, resulting in a framework that uses its own integration features in constructing itself, i.e. it has to some extent been bootstrapped.

The UniForM Workbench is an *integration framework*, i.e. a software system that provides a skeleton software architecture in terms of appropriate Haskell class definitions, complemented with a library of integration services. The skeleton architecture provides the features common to a family of applications and can be specialized for creating a specific application. The integration framework

provides in turn an extensible collection of services addressing control, data and tool integration issues.

2 The Z-WorkBench

The UniForM Workbench is a generic architecture. A particular development environment can be integrated on top of this basis, which provides the specialized tools to support the development life-cycle.

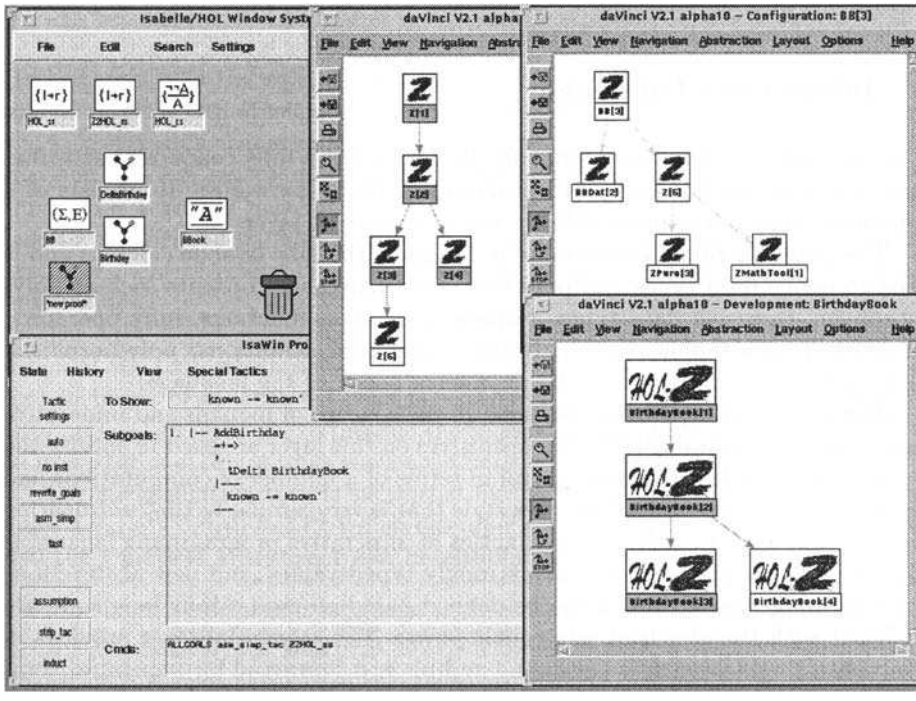


Fig. 2. Z-Workbench

One such instance is the Z-Workbench [31,30] - a SDE for formulating and proving Z [49] specifications. The main component of the Z-Workbench is the encoding of Z into Isabelle [39] called HOL-Z [22]. HOL-Z reads Z specifications and type-checks them. One can then prove theorems within the specification, generate formatted documentation, or if the specification can be shown to be executable, generate program code. A text editor (presently, the standard OpenWindows editor **textedit**) can be used to edit specifications.

A session with the Z-Workbench is pictured in Figure 2 (from [30]). On the right, we can see daVinci visualizations of the development and configuration graphs, and on the left HOL-Z' graphical user interface [21]. In the bottom

right corner is the *development graph* of the `BirthdayBook` specification. The numbers `[1]`,..., `[4]` denote the different versions; the edges of the graph denote the development relation. The *configuration graph* of version `[4]`. shows the objects comprising the development in the upper right corner. This configuration imports version `Z[5]` of the standard Z library, whose *version graph* is in turn shown to the left. The development graph denotes the development of entire configurations using HOL-Z. HOL-Z is running under Standard ML (SML) [33], and each development is actually a persistent "dump" of the corresponding SML system state. The various proof obligations are attached to this session object, and stored on the level of the persistent store as individual objects in order to facilitate formal development in the many.

3 Integration Language

The lazy and pure functional language Haskell [15] is a fully concurrent scripting and programming language with programs as first class values. It consists of a functional kernel, a sequential layer and a concurrent layer.

The pure *functional kernel* takes its origin in the lazy lambda calculus and is used to define data types and pure functions. Language concepts such as polymorphism, recursive data types, laziness, first class functions, infix operators, currying, partial application, patterns, equational definitions, polymorphism, and modules all contribute to the expressive power of the language.

The *sequential layer* takes its origin in the concept of monads and imperative functional programming [43]. The primitives on this layer are used to specify the behaviour of single threads of control in terms of a number of computations and combinators over computations, in a style that is very similar to the *continuation passing style* used to define the semantics of imperative programming languages [4]. Program scripts are first class, strongly typed values, and new abstractions for expressing flow of control can be defined based on need. Many languages are designed with the idea that size equals power. The sequential layer provides in contrast a single `return v` command and an `a;c` command for sequential composition of an action `a` with a continuation `c`. This kernel can then be extended on demand with combinators for selection, iteration and sequential composition.

The *concurrent layer* has its semantic background in Concurrent Haskell [40] and is used to define the interaction between the individual threads of control in terms of shared memory abstractions. An extension of this basis in the form of the UniForM Concurrency Toolkit [36, 18] provides a message passing model similar to the one of Concurrent ML (CML) [45] offering selective communication over channels, i.e. a communication model close to *process algebras* such as CCS [32] and the (higher order) π-calculus [44, 47].

There are several reasons why this language framework is extremely suitable for tool integration within the context of SDE's for formal or rigorous program development. Functional languages are, due to their higher order language features and expressive power, very suitable for rapid prototyping (i.e. rapid integration). Moreover, they combine the merits of *formal specification techniques*

with that of *practical software development*, and due to the presence of a fully integrated interpreter and compiler [50, 16] together with extensive libraries, it is possible for a functional language to serve as an efficiently compiled *systems programming language*, or as an expediently interpreted *scripting language* at one and the same time. The language dichotomy frequently found in operating systems such as Unix (C versus csh) is consequently avoided. When programming in a functional language, the size of the program is usually significantly decreased and overall productivity increased, with about a factor 2-3 compared to traditional imperative languages (see e.g. [14]).

4 Integration Manager

Haskell may take the role of an *extensible language* on the one hand, and the role of an *embedded language* on the other. The UniForM Workbench is an open ended integration framework. Existing components, such as type checkers and theorem provers, can be made to cooperate by integrating them into the framework. This typically means that the tools are encapsulated and set up to interface to the Repository Manager (data integration), the Subsystem Interaction Manager (control integration) and the User Interaction Manager (presentation integration).

The first step towards such an integration is to provide a Haskell *encapsulation* of the tool, which defines the data, services, events and exceptions of the tool in terms of an abstract data type. Given these wrappers, Haskell is from then on used as *glueware* to provide the missing bits and pieces of code required to make the individual tools work together.

The encapsulation task is supported by the Integration Manager which provides a variety of utilities by which interoperability between foreign components and the UniForM Workbench can be established:

Green Card is a foreign language interface by the Glasgow Functional Programming Group [42], which can be used to wrap Haskell functions around existing C functions, thus providing an interoperability bridge to all kinds of existing "C-Ware".

Haskell-Expect is a utility for wrapping Haskell interfaces around loosely coupled, interactive Unix tools [17]. It allows the Haskell program to take control over the dialogue with the Unix tool, i.e to emulate user interactions, in the form of send command-receive response sequences. Haskell-Expect has been inspired by, but improves on, the `expect` [29] utility for Tcl.

CORBA/COM Gateway is a generic interoperability bridge, that will provide access to existing CORBA components through their interfaces defined in the Interface Definition Language (IDL). A COM Gateway is currently available from the Glasgow Functional Programming Group [41].

Scripting Interface allows foreign components to call services of the UniForM Workbench by passing it Haskell scripts in a raw textual form. The UniForM Workbench is required to run under the Hugs interpreter [16] if this feature is needed, since the client uses Haskell as an embedded language.

The first two components of the Integration Manager provide means by which Haskell interfaces can be wrapped around existing software in order to use their services within the UniForM Workbench. HOL-Z has for example been integrated using Haskell-Expect, allowing the workbench to take control over the session with SML.

The last component allows the UniForM Workbench to be controlled from the outside, i.e. by a surrounding (say non-formal) SDE. The CORBA gateway, planned for the future, will in principle provide interoperability in both directions: a Haskell client may call servers developed in foreign languages such as C(++), Perl and Java (among others), whereas clients in other languages can use CORBA to access the services of the UniForM Workbench. A large variety of tools can consequently be integrated using the Integration Manager.

5 Subsystem Interaction Manager

An SDE is basically a reactive, *event driven* system with events amounting to user interactions with the User Interaction Manager, change notifications from the Repository Manager, individual tool events, as well as operating system events. The central component in this reactive system architecture is the Subsystem Interaction Manager which takes the role of the *control component*, whereas the individual tool components form the *environment* of the reactive system. The external tools are controlled through a number of abstract data types constructed with help of the Integration Manager as described in the previous section. The Subsystem Interaction Manager consists in turn of a network of concurrent communicating agents (also called *interactors* or *event listeners*).

Users interact with the repository on a "query by navigation" basis, through different kinds of filtered views such as version, development, and configuration graphs that have been set up using the User Interaction Manager. These views are constructed according to the Model-View-Controller paradigm [23], with the Repository Manager in the role of the *model*, the User Interaction Manager in the role of the *views* and interactors of the Subsystem Interaction Manager in the role of the *controller* (see Figure 3).

The Repository Manager provides interoperability between tools by letting them store and exchange software objects via the persistent store. The system has been designed to work in a multiuser setting, where each user is interacting with the system using his own instance of the UniForM Workbench. When one user issues a request to create a new revision using the Z-Workbench, say, this request is delegated onwards from the User Interaction Manager to the interactor listening to the request. This interactor will create a new version in response by sending a request to the Repository Manager, and will also start up a text editor for editing the associated Z specification. The notification manager of the Repository Manager will, as soon as the new revision has been created, broadcast change notifications to all running workbenches informing them that a new revision has appeared. Each workbench may have interactors set up to listen to such events, and they will typically respond by sending requests to

the User Interaction Manager that the associated views, i.e. the version graph, should be modified accordingly to render the new version.

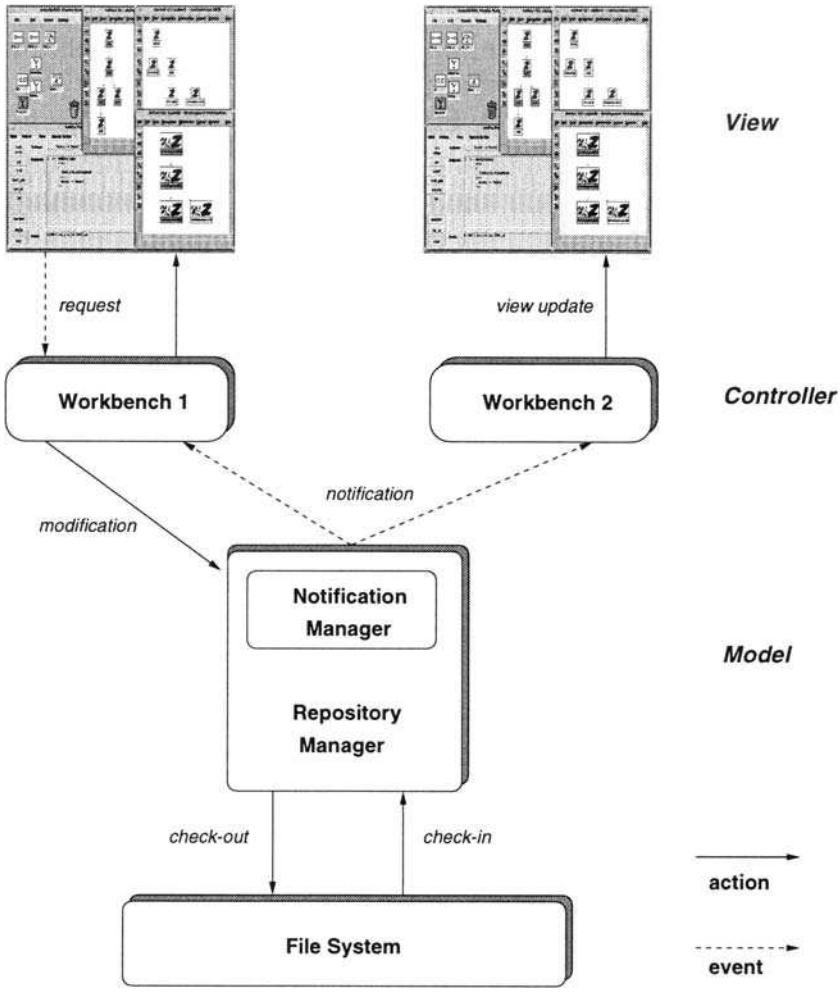


Fig. 3. Reactive System Architecture

The workbench represents, in the style of CML, events in terms of abstract *event values* which denote communications. Some base events of the Subsystem Interaction Manager amount to *internal communications* over typed channels, in the form of a `send ch v` event and a `receive ch` event for sending and receiving values over the channel `ch`, respectively. Other event values are then used to represent *external events* of the environment, such as the event `signaled`

`s` that occurs whenever the operating system signal `s` is generated. Additional event values are in turn used to denote user interactions, repository change notifications, tool terminations, and tool responses.

The approach to event handling is fully composable. New composite events can be defined from existing ones by applying the *guarded choice* operator (`e1 +> e2`) corresponding to the sum operator of process algebras, or the *event-action* combinator (`e >>>= f`) that combines an event `e` with some additional *reactive behaviour* in terms of the continuation function `f`. Abstraction and information hiding is therefore fully supported by the approach.

The Subsystem Interaction Manager operates with two kind of domains: *events* that are generated by the environment and *actions* that define the response to an event. Actions are represented in terms of computations of type `IO a`, whereas events of the environment are represented in terms of values of type `IA a`. Both are functors parameterized over the value returned when the action has been performed or when the event occurs.

The approach is inspired by, but improves on existing approaches such as Toolbus [3]. It uses *adaptors* as central components, that are interposed between the external event generating tools and the event listeners. It is the role of the adaptors to turn external events, independent of their physical representation, into first class event values, and to delegate such events, whenever they occur, to the relevant event listeners of the application. The *event listeners* are, on the other hand, concurrently executing threads of control defining the logic of the application by synchronizing on event values. A new interactor that repeatedly listens to the events defined by `e`, can be created by calling the `interactor e` command. The interaction pattern can be changed later on to `e'` by calling `become e'`. The interactor model has been inspired by the actor model [1], and provides a concept of *iterative choice*.

An integrated SDE is then typically structured in terms of a number of concurrently executing interactors that are set up to react to the events of the environments as well as the internal events of the system. The *interaction pattern* of an interactor is usually defined as a guarded choice, and the interactor is set up to execute an action in response to each event:

```
interactor (\iact ->
      e1   >>>= \v -> do {f1 iact v}
  +>  e2   >>>= \v -> do {f2 iact v}
  ...
  +>  en   >>>  stop iact
  )
```

An interactor can be set up to solve a single task of the application, such as monitoring user interactions and repository change notifications related to a single window, e.g a version or a configuration graph. This approach leads to better locality, and thus improved application structure. The interactor will automatically register interest in the relevant events, and it will, likewise, deregister interest in such events should the interactor terminate [19]. In traditional

approaches to event handling based on callbacks, the application must explicitly deal with those mundane technicalities.

The concepts of interactors and first class event values define a more functional and agent based approach to event handling compared to traditional paradigms based on callbacks. Callbacks are essentially procedures returning nothing of interest, whereas event values are functors that can be mapped to return some other value when the event occurs using the event-action combinator. Interactors can in turn be used to emulate more traditional approaches such as event loops, event handlers (callbacks) and the model-view-controller paradigm. Nothing is therefore taken away from a novice system integrator, but a lot is gained for the competent one.

6 Repository Manager

The Repository Manager [20, 52] serves as a *persistent store* for the software objects that may occur within a formal, rigorous or structured software development environment: specifications, proof obligations, proofs, tests etc. It also records the various relationships between such objects, such as *development links* (revisions, refinements), *dependency links* (imports) or plain *reference links*.

The Repository Manager has been built on top of H-PCTE [7], a public domain implementation of the Portable Common Tool Environment (PCTE) [10]. It consists, at the time of writing, of the Object Management System, the Notification Manager and a Check-in/Check-out utility. Prototypical features in support of version and configuration management are given as well.

The *Object Management System* (OMS) provides a persistent store for attributed graph structures with typical database features in support of transactions and schema definitions. It extends a usual database with *long field attributes*, i.e. files, needed for SDE's and multimedia databases, as well as persistency by reachability, which fits nicely into a functional framework since it is based on the garbage collection paradigm: objects not reachable from the persistent root are automatically garbage collected. The repository can be viewed as a web of software objects. Attributes are used to describe *properties* of these objects, such as the name of the object and its development status. Links are furthermore used to define *relationships* between the objects, such as traces between specifications and their associated proof obligations.

The *Notification Manager* provides the application with means to listen to *change notifications* whenever the repository is modified. In other words, the Repository Manager is an active database. The change notifications, signaling attribute modifications (`modified o anm`), object/link deletion (`destroyed o`) and link creation (`appended o`), are all represented in terms of first class composable events of type `IA` that can be handled by interactors. Interactors can be set up to await specific events to occur, such as for example the event that occurs when an attribute associated with a persistent object is modified. This feature has been used to construct *dynamic views* over the repository, so that changes made in one view are automatically reflected in all other relevant views

[20]. Being based on notifications, view update happens asynchronously. This is an important feature for a formal SDE, sinced it allows a developer to get notified immediately whenever some relevant proof has been verified (or rejected) by someone else.

The *Check-in/Check-out* utility is provided in the form of a toolkit, by which the "long field" attributes of persistent objects can be checked-out to the file system and checked-in again after they have been developed using tools such as editors, theorem provers and model checkers. Files are stored within the repository as well, in order to maintain the ACID (Atomicity, Consistency, Integrity, Durability) property of transactions for objects as well as for files. This kind of *workspace* model provides a simple concept of long design transactions.

Haskell serves as a *query, data definition* and *data manipulation language* in this context. Any Haskell value that can be given a textual representation can be used to describe properties of objects. Currently, there is however no support for orthogonal persistence which prohibits the storage of higher order values. Queries are formulated as Haskell computations, but by exploiting higher order imperative programming we reach a level of abstraction that lies somewhere in between that of 4GL database programming languages used for tuple at a time manipulation on the one hand, and SQL for declarative queries on the other.

Data integration typically goes beyond integration using a shared repository of persistent objects. Frequently, the output generated by one tool, must be prepared for input by another tool. This leads to the requirements for a *common data interchange format*. The UniForM Workbench does not insist that all objects are represented according to a single predefined formalism. For many cases an ad-hoc representation is most suitable, such as for example the Z-Workbench, which uses the email format [35] as a universal representation for Z specifications.

7 User Interaction Manager

In many cases, tools come with their own interface, and very little can be done to achieve presentation integration. Frequently, however, we would like to reuse tools that come with a command interface, and then wrap GUI's around these tools. Moreover, there is a need within an integrated environment to develop views over the repository such as version, configuration and development graphs.

The User Interaction Manager provides features that support exactly these tasks. The two most important components of the User Interaction Manager are Haskell-Tk and *daVinci*. Haskell-Tk is a concurrent, strongly typed and higher order encapsulation of Tk [38] that supports the definition of *user dialogues*. Haskell-Tk is also the tool to use when one wants to wrap graphical user interfaces around existing non-graphic tools. *daVinci*, on the other hand, is a *graph visualization* system [12] used to construct graph oriented, fully interactive views such as version, configuration and process graphs. *daVinci* interfaces in turn to Haskell-Tk, which is used in this context to define the dialogue windows and input forms.

The User Interaction Manager uses concurrently executing interactors to handle user interactions. All kinds of *user interactions* such as mouse button clicks, keyboard presses, and menu invocations are defined in terms of event values of type `IA`. The base event for listening to user interactions, is the event `userinteraction o e`, where `o` denotes a graphical object and `e` the event pattern defining the user interaction. All other events of Haskell-Tk are defined as abstractions over this base event using the guarded choice and the event-action combinator. A menu invocation event is for example defined as a guarded choice over the constituent menu item selection events.

The User Interaction Manager is a strongly typed, higher order system making use of Haskell type classes. Just as in object oriented GUI's, the major bulk of functionality is laid out by an extensive *class hierarchy* [51]. The individual widgets such as buttons and menus, are then established by straightforward instantiations of this class hierarchy. This feature, combined with the fact that event values are fully composable, and that most of the widgets are polymorphic, means that new composite widgets offering functionality at a higher level of abstraction can be developed from existing ones.

This feature has been used to provide a *toolkit* that takes the expressive power far beyond the one of the Tk toolkit. Some archetypical user interface components of SDE's, such as input forms for editing the attributes of these objects, can therefore be developed as reusable, generic ADT's. When developing SDE's, one wants to spend as little time as possible with the details of the user interface. The User Interaction Manager has been designed with this in mind. Practical experience using the User Interaction Manager has shown that the typical higher order features lead to an improved application structure. Archetypical problems of traditional GUI's, such as those reported by Meyers [34] in "Eliminating the Spaghetti of Callbacks" are therefore not noticed in a carefully designed higher order, fully concurrent setting.

8 Conclusion and Future Work

Provision of concurrency, GUI's and gateways to foreign systems within a functional setting have been a lively track of research within the past years. The UniForM Workbench goes one step further by providing an integrated environment, which in addition deals with issues related to the integration of an active database. Haskell serves in this context as a uniform systems programming and scripting language. Moreover, it serves (very successfully) as an extension language, an embeddable language and a coordination language.

Haskell, together with the various extensions and abstract services of the UniForM Workbench, have demonstrated to provide a viable approach to tool integration. Above all, we have succeed in establishing a set of generic services of the integration framework that reduce the efforts that has to be invested in coming up with instantiations such as the Z-Workbench. A key feature in achieving this is the higher order approach to event handling combined with having program scripts as first class values.

The most complicated application of the integration framework is actually the UniForM Workbench itself, which is partially developed using its own services. The workbench consists of approximately 50K LOC of Haskell, which makes it the second largest known application of Haskell around the world. The Z-Workbench is a much simpler application, which is developed with approximately 5K LOC of Haskell.

Future work will mainly focus on 3 areas. One is the provision of the CORBA Gateway which will, technically, take a starting point in the existing COM binding. The second, and major track of future research, concerns the Repository Manager and the provision of generic and fully fledged version, configuration, workspace and process management utilities (what has been done so far in this direction can be viewed as tentative work). As a third issue more applications will be developed, either in the form of instantiations of the integration framework such as a workbench for CSP specifications, or in the form of generic integration services such as a database schema editor.

Acknowledgement

This work would not have been possible without the achievements of the Glasgow Functional Programming Group and the consistent support of Prof. Bernd Krieg-Brückner at Universität Bremen. I would furthermore like to thank Walter Norzel for implementing selective communication and Stefan Westmeier for providing the encapsulation of H-PCTE. Carla Blanck Purper implemented parts of the lower-level communication with daVinci, which was in turn provided by Michael Fröhlich and Mattias Werner. Finally, the Z-Workbench is joint work with Christoph Lüth, Kolyang, Stefan Westmeier, Burkhart Wolff and the author.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, Massachusetts, 1986.
2. R. Bahlke and G. Snelting. The PSG System: From Formal Language Definitions to Interactive Programming Environments. *ACM Transactions on Programming Languages and Systems*, October 1986.
3. J. A. Bergstra and P. Klint. The Toolbus - a Component Interconnection Architecture. Technical Report P9408, University of Amsterdam, 1994.
4. D. Bjørner and C. B. Jones. *Formal Specification and Software Development.* International Series in Computer Science. Prentice-Hall, 1982.
5. B. Borras and D. Clement. CENTAUR: the system. *ACM SIGPLAN notices*, 24(2), 1989.
6. Microsoft Corporation. The Component Object Model Specification, 1998.
7. The H-PCTE Crew. H-PCTE vs. PCTE, version 2.8. Technical report, Universität Siegen, June 1996.
8. A. Deursen, J. Heering, and P. Kling, editors. *Language Prototyping - An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing.* World Scientific, 1996.

9. ECMA. *Reference Model for Frameworks of Software Engineering Environments*, 3 edition, June 1993.
10. ECMA. *Portable Common Tool Environment (PCTE) — Abstract Specification.* European Computer Manufacturers Association, 3 edition, December 1994. Standard ECMA-149.
11. Open Software Foundation. *OSF/Motif Series.* Prentice Hall, 1992.
12. M. Fröhlich and M. Werner. daVinci V2.0.3 Online Documentation. Universität Bremen, `http://www.informatik.uni-bremen.de/~davinci`, 1997.
13. A.N. Habermann and D. Notkin. Gandalf: Software Development Environments. *IEEE Transactions on Software Engineering*, December 1985.
14. P. Hudak and M. P. Jones. Haskell vs. Ada vs. C++ vs. Awk vs. ... - An Experiment in Software Prototyping Productivity. Technical report, Department of Computer Science, Yale University, July 1994.
15. P. Hudak, S. L. Peyton Jones, and P. Wadler. Report on the Programming Language Haskell - a non strict purely functional language, version 1.2. *ACM SIGPLAN notices*, 27(5):1–162, 1992.
16. M. P. Jones and J. C. Peterson. *Hugs 1.4, The Nottingham and Yale Haskell User's System*, April 1997.
17. E. W. Karlsen. Integrating Interactive Tools using Concurrent Haskell and Synchronous Events. In *CLaPF'97: 2nd Latin-American Conference on Functional Programming*, September 1997.
18. E. W. Karlsen. The UniForM Concurrency Toolkit and its Extensions to Concurrent Haskell. In *GWFP'97: Glasgow Workshop on Functional Programming*, September 1997.
19. E. W. Karlsen. *Tool Integration in a Functional Setting.* Forthcoming dissertation, Universität Bremen, August 1998.
20. E. W. Karlsen and S. Westmeier. Using Concurrent Haskell to Develop Views over an Active Repository. In *IFL'97: Implementation of Functional Languages*, September 1997.
21. Kolyang, C. Lüth, T. Meyer, and B. Wolff. TAS and IsaWin: Generic Interfaces for Transformational Program Development and Theorem Proving. In *Proc. TAPSOFT'97*, volume 1214 of *LNCS*. Springer Verlag, 1997.
22. Kolyang, T. Santen, and B. Wolff. A Structure Preserving Encoding of Z in Isabelle/HOL. In *1996 International Conference on Theorem Proving in Higher Order Logic*, volume 1125 of *LNCS*. Springer Verlag, 1996.
23. G. Krasner and S. Pope. A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.
24. B. Krieg-Brückner. UniForM Perspectives for Formal Methods. In *International Workshop on Current Trends in Applied Formal Methods*, October 1998.
25. B. Krieg-Brückner and B. Hoffmann, editors. *Program Development by Specification and Transformation: The PROSPECTRA Methodology, Language Family and System*, volume 680 of *LNCS*. Springer Verlag, 1992.
26. B. Krieg-Brückner, J. Peleska, E. R. Olderog, D. Balzer, and A. Baer. Universal Formal Methods Workbench. In U. Grote and G. Wolf, editors, *Statusseminar des BMBF: Softwaretechnologie.* Deutsche Forshungsanstalt für Luft- und Raumfahrt, Berlin, 1996. Available at `http://www.informatik.uni-bremen.de/uniform`.
27. M. Lacroix and M. Vanhoedenaghe, editors. *Tool Integration in an Open Environment*, volume 387 of *LNCS*. Springer Verlag, 1989.
28. C. Lewerentz. Extended Programming in the Large in Software Development Environments. *ACM SIGPLAN Notices*, 24(2), February 1989.

29. D. Libes. expect: Scripts for controlling interactive processes. In *Computing Systems, Vol 4, No. 2*, Spring 1991.
30. C. Lüth, E. W. Karlsen, Kolyang, S. Westmeier, and B. Wolff. HOL-Z in the UniForM WorkBench - a Case Study in Tool Integration for Z. In *Proceedings of ZUM'98: 11th International Conference of Z Users*, September 1998.
31. C. Lüth, E. W. Karlsen, Kolyang, S. Westmeier, and B. Wolff. Tool Integration in the UniForM WorkBench. In *Proceedings of TOOLS'98: Workshop on Tool Support for System Specification, Development, and Verification*, June 1998.
32. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
33. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.
34. B. A. Myers. Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, November 1991.
35. J. Nicholls and The Z Standards Panel. Z notation, September 1995. Available at `http://www.comlab.ox.ac.uk/oucl/groups/zstandards/`.
36. W. Norzel. Building Abstractions for Concurrent Programming in Concurrent Haskell. Master thesis, Universität Bremen, April 1997.
37. OMG. The Common Object Request Broker: Architecture and Specification, Revision 2.0. Technical report, The Object Management Group, July 1995.
38. J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley, 1994.
39. L. Paulson, editor. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS.* Springer Verlag, 1995.
40. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Principles of Programming Languages '96 (POPL'96), Florida*, 1996.
41. S. Peyton Jones, E. Meijer, and D. Leijen. Scripting COM components in Haskell. *Submitted to Software Reuse*, 1998.
42. S. Peyton Jones, T. Nordin, and A. Reid. Green card: a Foreign-Language Interface for Haskell. In *Proc. Haskell Workshop*, 1997.
43. S. Peyton Jones and P. Wadler. Imperative Functional Programming. In *Proc. 20th ACM Symposium on Principles of Functional Programming*, 1993.
44. D. Walker R. Milner, J. Parrow. A calculus of mobile processes, part i. Department of Computer Science, University of Edinburgh, June 1989.
45. J. H. Reppy. *Higher-Order Concurrency.* PhD thesis, Department of Computer Science, Cornell University, 1992.
46. T. Reps. *Generating Language Based Environments.* PhD Thesis, Cornell University. MIT Press, 1983.
47. D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis, Department of Computer Science, The University of Edinburgh, 1993. CST-99-93.
48. D. Schefström and G. van den Broek. *Tool Integration.* Wiley, 1993.
49. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1992. 2nd edition.
50. The GHC Team. *The Glorius Glasgow Haskell Compilation System, Version 2.10, User's Guide*, January 1998.
51. T. Vullings, W. Schulte, and T. Schwinn. The Design of a Functional GUI Library using Constructor Classes. In *Perspectives of System Informatics*, LNCS. Springer Verlag, 1996.
52. S. Westmeier. Verwaltung versionierter persistenter Objekte in der UniForM Workbench (UniForM OMS Toolkit). Master thesis, Universität Bremen, February 1998.

Two Real Formal Verification Experiences: ATM Switch Chip and Parallel Cache Protocol

Masahiro Fujita[1], Sree P. Rajan[2], and Alan Hu[2]

[1] Fujitsu Laboratories of America
595 Lawrence Expressway
Sunnyvale, CA 94086
+1-408-530-4505 (Tel.) +1-408-530-4518 (Fax.)
fujita@fla.fujitsu.com (E-mail)
[2] Dept. of Computer Science
University of British Columbia
Vancouver, BC V6T 1Z4, Canada

Abstract. In this paper, we report two of our recent efforts in applying formal verification methods to our real hardware designs. The first one is to try to verify ATM switch LSI chips through the combined use of a theorem prover and model checking programs, and the second one is to try to formally verify the correctness of a cache coherency protocol used in one of our parallel PC servers by model checking programs. In both cases, the verifications themselves were successful (we could really verify the "abstracted/simplified" designs). We could not, however, get much benefits from formal methods, since the verification process was not automatic but interactive. We had to spend significant amount of human time and human efforts in applying formal verification techniques, which made it very difficult to verify designs "in time", that is, before the design process finishes. We review our experiences and describe problems that we typically encounter in application of formal verification techniques to real life designs.

1 Introduction

Formal verification has been getting more and more attention even in industries. In fact, we, Fujitsu, have been using formal verification technology as early as in the beginning of 80's [MF85]. Although those initial attempts got some success, they did not become part of the main design methodology in Fujitsu, mainly because designs in RTL were not common in those days. Also, the idea of formal verification was not well understood by typical designers. They did not like to spend lots of time in understanding what is formal verification and how to efficiently use formal verification tools.

But now the situations have been completely changed. Because of the high design complexity and because of the high pressure on time-to-market, designers are willing to spend some of their time for formally verifying their designs. They

now realize that just simulating their designs as much as possible may not work. They are now trying to understand how formal verification technology can be used for which part of their designs. Along this direction, we have had several trial verification of actual designs.

In this paper, we report two of such attempts; one is on the verification of ATM switch LSI chips with a theorem prover as well as model checking programs, and the other is on the verification of the correctness of a cache coherency protocol for parallel PC servers. In both cases, our researchers, instead of designers, actually verified the designs. This is because we need a lot of experiences and good understandings on the formal verification tools in order to effectively use them. This is clearly one of the major problems in application of formal methods. But our hope is that if verification researchers clarify in which way we can effectively and efficiently use formal verification tools, at least some of real designers can start to use formal verification tools in some of their actual designs.

In the following, we report our verification attempts from a high level viewpoint. That is, instead of describing technical details, we review the entire verification processes and point out exactly what are the problems in the application of formal verification techniques to real life designs.

2 First example: ATM switch LSI chip verification

2.1 What is ATM switch ?

Asynchronous Transfer Mode (ATM) has emerged as a backbone for high-speed broadband communications networks [CFFT96]. An ATM network backbone typically consists of a number of small ATM switches interconnected in a matrix topology. An ATM switch takes data from input ports and forwards the input data to the proper output ports in the same order as the input data. An ATM switch is typically designed as a RAM-embedded Application Specific Integrated Circuit (ASIC). Because of various applications of ATM switch as the core component of various ATM networks, short product life-cycle and changing standards demand an efficient methodology to modify the current design for a future product.

An ATM switch forms the basic component of an ATM network, in which several switches could be connected in the form of a matrix for larger bandwidth. The modeling and functionality can be illustrated by a *small capacity ATM switch*, whose typical architecture is shown in Figure 1. The ATM switch has 2 input ports, 2 expanded ports, and 2 output ports. ATM cells from the network arriving at the input ports `iHW0` and `iHW1` are collected by the `INPUT MODULE`. The cells are forwarded to the `CELL PROCESSING MODULE`, which stores the cells in the proper addresses and routes them to the corresponding output ports through the `OUTPUT MODULE`.

The internal switch clock runs slower than the external network clock in order to reduce the cost of the different components within the ATM switch.

Therefore, a serial-to-parallel (S/P) conversion module, as part of the **INPUT MODULE**, buffers the incoming words for processing multiple words in the buffer simultaneously. The S/P conversion ensures that the switching speed is maintained at the the original external network clock (e.g., 156MHz), notwithstanding switch clock speed reduction. The Write-Control (WC) unit belonging to the **CELL PROCESSING MODULE** obtains addresses from the Write-Address-FIFO (WAF) unit and stores the incoming cells from the S/P unit in the addresses obtained from WAF. The WAF, while providing the addresses to the WC, also provides the same addresses matching the corresponding cell TAGs to one of the two Read-Address-FIFOs: RAF0 and RAF1 depending on which output port (port 0 or 1) the cell should be switched. A cell can also be switched to more than one output port depending on the header information. In this case we call this cell the "copy cell" and a copy cell flag will be set. The expanded input ports, exHW0 and exHW1, send a Read-Request signal to the Read-Control (RC), which then obtains the addresses from RAF0/RAF1 and reads out the corresponding cells from the cell RAM. A parallel-to-serial (P/S) conversion in the **OUTPUT MODULE** serializes the cell words to match the faster external network clock at the output ports. The address of the outgoing cell is recycled back to the WAF unit to be reused for the next incoming cell.

When power is set, since there is no address in WAF nor RAFs, original addresses are generated by wBC(cell counter). After all addresses for Cell Buffer (cell FIFO) are generated, wBC is detached and WAF and RAFs are connected to form address recycle loop. If two RAFs (RAF#1/RAF#0) are empty (that is, Cell Buffer = empty), the system comes to "Empty-reset" state. This state is equal to "power setting" state, namely, all addresses in WAF/RAFs are reset. New addresses are generated again by wBC.

2.2 Debugging by a model checker

Here we present our first verification attempt on ATM switch designs. Actually it was not a verification but a debugging process [CYF94].

During a field test, the following abnormal behavior appeared on the chip. That is, after several seconds of operations, some data from oHW0/oHW1 disappeared while some other data ware duplicated. Since there are several seconds before the abnormal behavior appears, it is impossible to realize the same situation by simulation; it needs billions of cycles of simulation (156MHz * several seconds). Instead we used symbolic model checking program, SMV [McM93], to model the abnormal behavior we observed.

We built three different models for model checking. Originally we suspected that FIFOs were not working correctly. So the first model we built is a detailed model for just FIFO and we intensively model checked it. However, we could not find any bugs, and so, we tried to build a model for the entire address recycle loop in order to capture the abnormal behavior. Since the size of hardware for the address recycle loop is too large for model checking, we concentrated only on control part. It has only 23 state variables and can be easily processed by

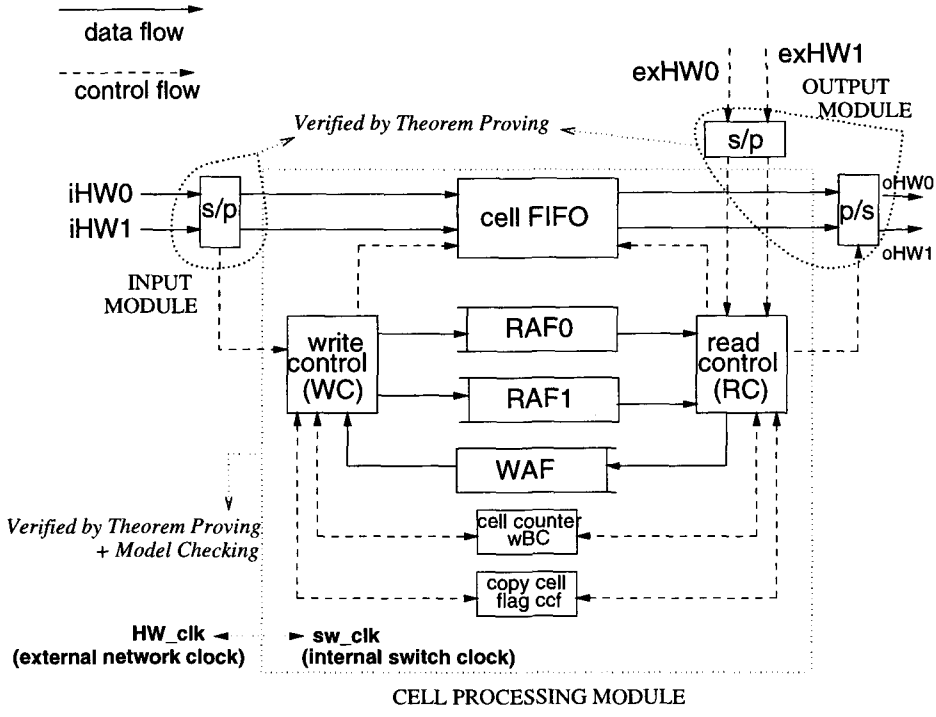


Fig. 1. Architecture of ATM switch with 2 input and 2 output ports

symbolic model checking programs. However, since it does not include datapath, we cannot describe the abnormal behavior directly. Instead we just checked various properties that must be satisfied by the hardware. This was somehow time-consuming process, but finally we found one property that gives some hints about the bug. We believed that when the system is reset, all hardware are reset "immediately". However, by examining a counterexample trace generated by model checking programs, we found that some of FIFOs need more than one cycle to be reset. This could be a source of bug.

So, we built the third model that has partial datapath on the FIFO which needs more than one cycle to be reset. With that model, we could specify the abnormal behavior and the model checking program generated a counterexample that illustrated why some incoming data can be lost. The bug was that because some part of hardware need more than one cycle to reset, some of originally existing addresses in the address recycle loop were not deleted. Since all addresses were regenerated again after reset, there were some duplication of addresses in the address recycle loop, and that was the reason why some incoming data were lost whereas some others were duplicated.

The counterexample that the model checking programs generated has more than 50 cycles. It could need intensive simulation to realize the same situation.

The above entire debugging process took two months with one full time working verification engineer.

2.3 Verification of a family of ATM switch by a theorem prover and model checking programs

Based on the above experiences, we have decided that ATM switch designs be formally verified. Since we have to design varieties of ATM switches, it is better to describe those ATM switches in a parametric way and verify all of them. So, we used a theorem prover and interactively proved them (once we find an appropriate strategy to guide theorem provers, our hope is that most parts of verification processes can be automatic).

We used a parametric model of an ATM switch in which

- The ratio of the internal switch clock to the external network clock is a generic value. The generic value can be instantiated with specific ratio numbers to produce corresponding switch models, which can then be piped into hardware synthesis.
- The number of input/output ports is a generic value. The generic value can be instantiated with specific numbers of ports to produce corresponding switch models. The specific switch models can then be input to hardware synthesis program.
- The buffer sizes of the ATM cell address FIFO units are generic values.

The parametric model is validated by formal verification of the properties using Prototype Verification System (PVS) [ORR+96]. PVS provides an integrated environment for the development and analysis of formal specifications and has a powerful theorem prover with a high-degree of automation together with a Binary Decision Diagram (BDD)-based model checker.

The goal of formal verification of the ATM switch model is to show the overall functional correctness property that *the input cells are switched to the proper output ports, and that the order of input cells is maintained upon output.* The overall correctness property subsumes the properties that input cells are not lost. Furthermore, it should be noted that the overall correctness property for the parametric ATM switch model should hold for any arbitrary clock ratio, an arbitrary number of ports, and an arbitrary buffer size of FIFOs.

The parametric ATM switch model is verified by first verifying the properties of its components. The verified component properties are then composed to show the overall correctness property. The components correspond to the processes of the high-level model. The model is first translated into the PVS specification language [ORR+96]. Verification of the properties is performed by invoking appropriate built-in proof strategies, in particular a combination of induction, rewriting, and special-purpose decision procedures for linear arithmetic and model checking using BDDs provides a semi-automatic verification of the parametric ATM switch model.

There are essentially 3 modules in the ATM switch: input serial-to-parallel conversion (S/P), cell processing or address supply/recycling (AR), and output parallel-to-serial conversion (P/S). The overall correctness property can be shown by composing the correctness properties of each of the 3 components.

The correctness property of each component is verified by further decomposing each property into sub-properties which can be proved with minimal prover guidance. Here we show part of the verification of cell processing or address supply/recycling (AR), and also the verification on the correctness of the entire ATM switch.

Formal Verification of the Address Supply/Recycling (AR) Module The address recycling module consists of the WC, the WAF, the RC, and one RAF for each output port. In the parametric model, the sizes of the WAF and RAFs are arbitrary. The corresponding PVS model of the recycling module is a state machine parametrized with respect to the buffer-size of WAF/RAFs/cell-RAM and number of input/output ports. The property that we would like to verify here is that the cell contents of the address recycled from RAFs to WAF should be processed by the output module prior to recycling the address. The correctness of recycling depends on whether the concurrent processes WAF, RAF, WC, and RC, correctly update the shared signals corresponding to `RAF_hptrs`, `RAF_tptrs` and `WAF_hptr`, `WAF_tptr`.

We can ensure the safety of address recycling process by checking that neither WAF head and tail pointers "cross" each other at any time: `WAF_hptr` does not cross `WAF_tptr` nor RAF head and tail pointers "cross" each other at any time: `RAF_tptr` does not cross `RAF_hptr`. Focusing on the head and tail pointers of WAF/RAF suggests that we can abstract the WAF/RAF buffers to each of size just 2: as shown in Figure 3.

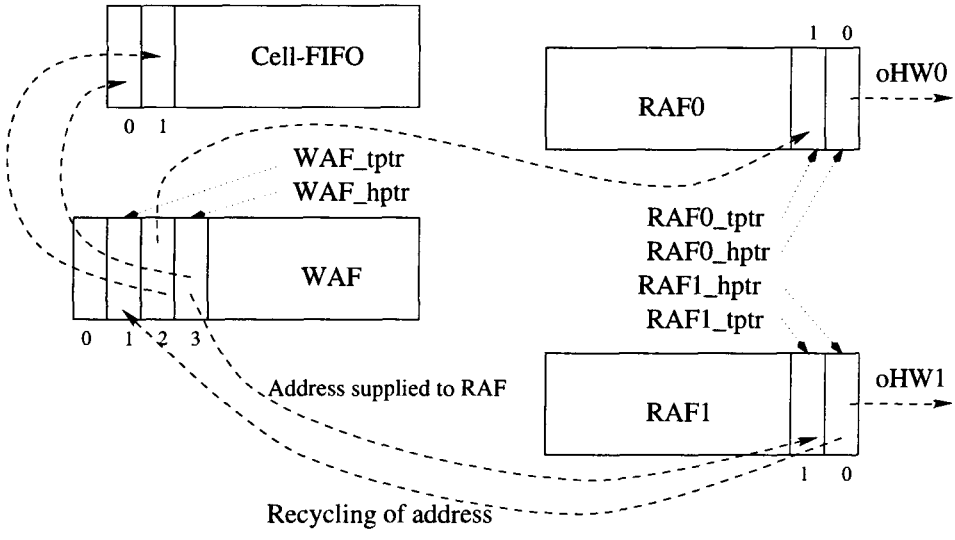


Fig. 2. Address Supply/Recycling in a 2-port ATM switch with generic buffer size for WAF, cell-RAM, RAF0/1

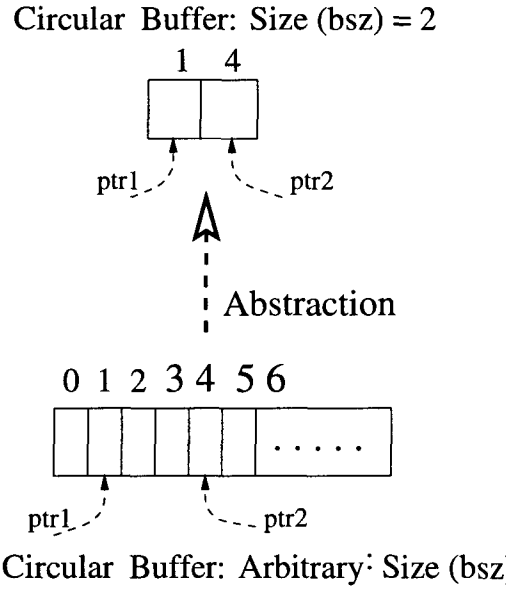


Fig. 3. Abstraction of circular buffer of arbitrary size to that of size 2: locations $\leq$ `ptr1` mapped to `ptr1` and the rest to `ptr2`. $\leq$ is evaluated prior to computing modulus (the buffer size) to avoid wrap-around the buffer.

Although this abstraction itself must be found by verification researchers/engineers, the correctness of it with respect to the safety property that pointers `ptr1` and `ptr2` never point to the same buffer location can be formally proven in PVS by simple arithmetic inequality reasoning. This is an essential advantage of using theorem provers as well as model checking programs. Because of this, it is sufficient to prove the case of Figure 3 in order to ensure the correctness of the original (non-abstracted) design.

In the PVS state machine model, WAF and RAF buffer sizes are arbitrary (uninterpreted) and the number of ports is also arbitrary. Thus we first instantiate the model with

- buffer size of 2 and
- number of input/output ports to 2

and verify the property that `WAF_tptr` and `WAF_hptr` never point to the same location in WAF, and also that `RAF_hptr` and `RAF_tptr` never point to the same location in RAF using BDD-based CTL model checking in PVS. However, the verification did *not succeed* for our original model, and produced a counterexample in the form of a series of unprovable sub-properties. A closer examination of the sub-properties **revealed bugs** that the pointers to the WAF locations (`WAF_hptr`, `WAF_tptr`), were not initialized, and were not incremented at every clock step. The model was updated and the property was model checked to hold

in the model. The state-space is small for a buffer size of 2, and the run-time for the verification is under **20 seconds** on Sparc-20 with 64 MB.

Verification of Overall ATM Correctness Property We need to verify the overall correctness property that the cells are output in the same order as the input. By composing the correctness properties corresponding to the 2 modules: input serial-to-parallel conversion (S/P) and address supply/recycling (AR), we can infer that the cell word input at an input port eventually appears at an RAF output at a certain time.

The correctness of Parallel-to-Serial conversion (P/S) is similar to Serial-to-Parallel conversion (S/P). Furthermore, we can show that the order of the cells is maintained at the output by showing that the order is maintained in each of the 3 ATM modules. All of these verification are realized on top of a theorem prover, PVS.

In summary, we had to verify 17 sub-properties corresponding to the 3 ATM modules to arrive the overall correctness property.

2.4 Some remarks

Our work uses a pragmatic combination of theorem proving and model checking within a single framework, in conjunction with simulation for efficient verification of large industrial software/hardware designs. As part of future work, we plan to provide a general proof strategy that combines proof rules we have used in verifying the overall correctness property of the ATM switch. The proof strategy could be used in verifying other designs similar to an ATM switch. We are investigating use of formal specification and verification for design trade-off/analysis to determine an efficient partitioning of the processes in the high-level model, and trade-offs with respect to clock ratio, number of input/output ports, and buffer size. Thus, a design methodology based on high-level parametric modeling coupled with the introduction of formal verification, in conjunction with small-scale simulation early in the design cycle, drastically reduces system design cost and time-to-market.

On the negative side, definitely we cannot say the proposed verification methodology is easy to be applied to other types of real designs. Right now verification engineers who have understood the details of both designs and verification tools could effectively use it. Theorem provers are sufficiently powerful in mathematic sense, but not so powerful from users' viewpoint. Also, we need an easy-to-use HDL front-end for theorem provers. Because semantics of popular HDLs are not so simple, translation from HDLs into methematical logic can be extremely tricky.

3 Second example: A cache coherency protocol verification

This section describes our experience applying formal verification techniques to the cache coherence protocol for parallel PC servers [HFW97]. In recent years, several researchers have described the verification of cache coherence protocols to demonstrate the potential of formal verification. Here, we sought to quantify this potential by carefully tracking the effort and results of applying formal verification, rather than simply demonstrating that verification was possible.

3.1 What should be verified ?

Our PC server to be verified, HAL S1 System, is a multiprocessing computer that supports both shared-memory and message-passing paradigms. It consists of symmetric multiprocessing (SMP) nodes connected by our proprietary interconnecting network. Each node is a standard SMP set-up containing four Intel Pentium processors. The interconnect is based on two chips: the Mesh Coherence Unit (MCU) and the Router. The MCU is the network interface for each node, translating bus transactions within each node to network messages and vice-versa. Zero or more Routers can be cascaded in arbitrary topologies to form the network fabric.

Here, we are only concerned with the cache coherence protocol of the S1 System, simply because it is clearly impossible to verify actual implemenation of the protocol, such as gate-level circuits, due to large circuit sizes. Within each SMP node, the normal bus-based snooping protocol keeps the processor caches coherent. We will assume the snooping protocol works correctly. A directory-based cache coherence protocol maintains coherence between nodes. This protocol is the target of our verification effort.

Main memory is physically distributed among the nodes, but each processor views all memory as a global, flat address space. To achieve good performance, processors cache the memory lines they access, whether the memory is located locally in the same node as the processor or in some other node. The cache coherence protocol must deliver data to processors that request it and ensure that all cached copies of a memory line are coherent, producing a cache-coherent, non-uniform memory access (CC-NUMA) machine. The cache coherence protocol maintains a directory that indicates which nodes have cached copies of which memory lines. This directory is physically distributed among the nodes, just as main memory is.

For example, suppose a processor in node A needs to read a memory location that is physically in node B. Node A will send a message through the interconnect to node B, requesting the memory line. If the memory line is available, node B will send a reply containing the requested memory line back to node A and update its directory to indicate that node A has the line cached. On the other hand, suppose the memory line requested by A is actually cached dirty at another node C. In this case, node B knows from its directory that node C has the line,

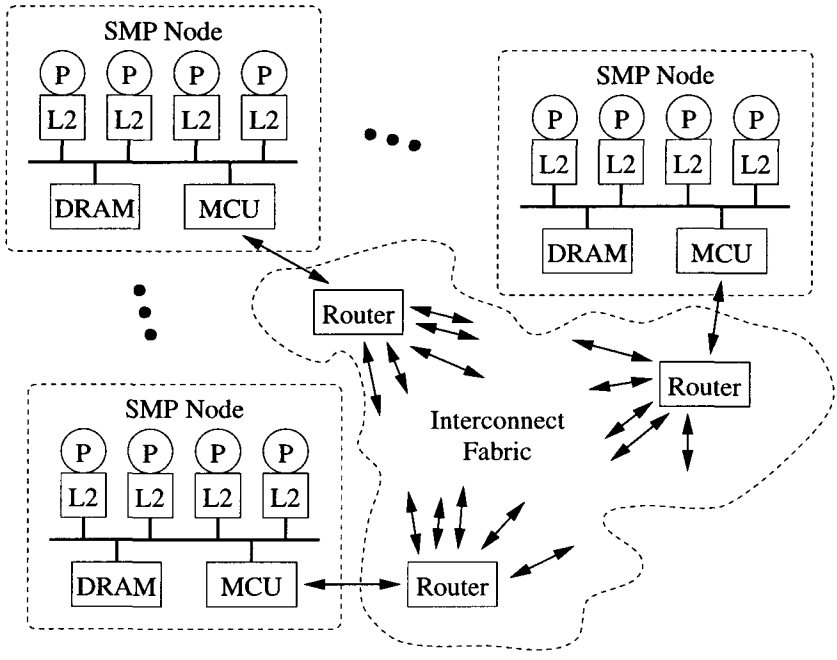


Fig. 4. Architecture of the HAL S1 System. Each node is a standard SMP configuration of Intel Pentium processors and local DRAM. The Mesh Coherence Unit (MCU) acts as a bridge between the internal bus of the SMP node and the network fabric. The current implementation supports up to 32 nodes (128 processors).

so it sends a message to node C asking for the memory line back or for C to forward the line to A. As before, B must update its directory accordingly.

The scenario becomes more complex when we consider that multiple transactions on the same memory line can be initiated simultaneously from different processors. The protocol must prevent race conditions and different transactions interfering with each other. In general, designing directory-based cache coherence protocols is a subtle art, making it an attractive target for formal verification.

3.2 Our approach

Few resources were committed to the formal verification project: basically, one person part-time. In our case, this level of commitment resulted from the confluence of a verification researcher looking to spend some of his time working on a real design as a case study and a design team with a complicated protocol that they felt would benefit from formal verification. In general, we expect a similarly low level of commitment to formal verification to be common in practice. For example, an architect, design engineer, or verification engineer might experiment with formal verification as just one of many responsibilities and as just one of many techniques to debug a complicated design.

Given this minimal level of commitment, we chose to try formal verification at the protocol-level — verifying the cache coherence protocol itself at a very high level, rather than verifying implementations of the protocol because protocol-level verification seemed feasible for one person working part-time.

However, our methodology in applying protocol-level formal verification was far from ideal:

- Ideally, protocol-level formal verification is done extremely early in the design cycle. In our case, we started protocol-level verification as RTL coding was nearing completion.
- Ideally, the designers/architects themselves use formal verification to explore preliminary ideas. In our case, an outsider came in to check a completed, mature design.
- Ideally, protocol-level formal verification catches bugs before much effort is spent on the design. In our case, any bugs found would require substantial effort to fix.

Unfortunately, we expect our scenario to be a common one in practice, so our experience with this less-than-ideal methodology is likely to be repeated by many others.

Previous experiences on protocol-level formal verification suggests that the following milestones occur in roughly this order:

1. Obvious Documentation/Specification Errors — Formal verification requires rereading the documentation from a fresh perspective, and many errors become immediately apparent.
2. Bugs in the Formal Model — Writing the formal model for verification is much like writing a small program. Simple coding errors (unrelated to bugs in the real design being verified) creep in and require a bit of debugging.
3. Subtle Documentation/Specification Errors — Once the formal model is reasonably debugged, the verifier will flag many "errors" in the design. At first, these are the result of misreading the design documentation, but at some point, the errors in the formal model result from ambiguous or erroneous documentation.
4. Bugs in the Real Design That Were Fixed Already — Since other debugging efforts are underway simultaneously with formal verification, and since incorporating design changes into the formal model takes some time, the first real bugs discovered by formal verification are likely to be bugs that were discovered already by other methods.
5. Newly Discovered Bugs in the Real Design — This is the payoff. Every bug caught earlier by formal verification saves considerable time, effort, and money that would have been spent fixing the bug later.

Our experiences matched this pattern closely.

We reached Milestone 1 in the second week of the project, after less than ten hours in total (See Table 1 below.) had been spent on formal verification. An

example of the obvious errors identified is that the documentation specified the wrong data structure to update in a certain situation.

We reached Milestone 3 in the seventh week of the project. An example of a subtle specification error discovered is that a certain state machine had eight possible states, named for example $a0$, $a1$, $a2$, $a3$, $b0$, $b1$, $b2$, and c. The protocol specification indicated a certain action if the state machine was in state "ax", but the correct specification was that the action be taken if the state machine was in any state other than c. This error is perhaps more intuitive if we explain that state c was a distinguished state, and that the a states and the b states were conceptually similar.

We reached Milestone 4 in the tenth week of the project. Over the course of the project, protocol-level formal verification discovered only one real bug in the protocol: a complex interaction among various subunits of the MCU chip would result in incorrect behavior when a node with dirty data wrote back the data, but the message crossed with a request for that node to forward the data to another node. This bug, however, had been caught and fixed five weeks earlier, but the person performing the verification had been working from documentation just over five weeks old.

We did not find any bugs with formal verification that had not been caught by other methods. This failure was due in part to the limitations of our methodology, but was also due to the maturity and stability of the protocol. Indeed, the cache coherence protocol was essentially the second implementation of a protocol that had been extensively scrutinized (including some formal verification) previously. Most of the bugs, therefore, were at the implementation level, which protocol-level verification cannot discover.

How much did the formal verification cost? Throughout the project, time spent on formal verification was carefully recorded. The most striking feature of the data in Table 1 is that almost no time was spent deciding how to model the problem. This situation resulted mainly from our ability to reuse the results of previous work on verifying directory-based cache coherence protocols. Apparently, enough work has been done on protocol-level verification of cache coherence protocols that the problem domain is well-understood and new projects can be done with minimal effort. Furthermore, reusing modeling decisions from previous, successful verification projects minimizes the likelihood of state-space explosion resulting from bad modeling decisions.

Later in the project, a great deal of time was spent waiting for the verifier to run. Obviously, a faster machine or a faster verifier would improve this problem. However, note that all of the interesting feedback from the verifier came before lengthy runtimes were required. These long verification runs might be worthwhile only if there are plenty of machines sitting idle.

Finally, a large fraction of time was spent in meetings. While time devoted to meetings is generally the target of derision, this time is actually crucial to the success of the verification project. Almost all of the time logged for meetings was from the regular status meetings of the group. Attending these meetings is necessary to keep the people doing formal verification up-to-date on the progress and any changes in the design. If a person is responsible for formal verification as

well as other aspects of the project, he or she would be attending these meetings anyway, so almost no additional meeting time would be required to support the formal verification.

3.3 Some remarks

Overall, we have shown that protocol-level formal verification of directory-based cache coherence protocols has become sufficiently well-understood to be reasonably straightforward. The protocol we verified was of representative complexity, and the time and resources required were demonstrably small.

Week	Overhead	Meetings	Study	Model	Code	Run	Total
1	2	1.25	0	0	0	0	3.25
2	2	6.75	3	0	0	0	11.75
3	2.75	8.75	4.75	0	0.5	0	16.75
4	2.75	9.25	5.5	0	1.75	0	19.25
5	2.75	9.75	8	0	3.25	0	23.75
6	4.75	11	8.25	0	4.25	0	28.25
7	4.75	11	8.25	0	8.25	0	32.25
8	4.75	12	8.25	0	13	0	38
9	4.75	16	8.25	0.5	15.75	0	45.25
10	4.75	18.75	8.25	0.5	20	0	52.25
11	4.75	18.75	8.25	0.5	22	3.25	57.5
12	4.75	19.75	8.25	0.5	22	14	69.25
13	4.75	19.75	8.25	1	22	19.75	75.5
14	7.75	26.25	8.25	1	24.5	19.75	87.5
15	7.75	26.75	8.25	1	24.5	22.25	90.5
16	7.75	26.75	8.25	1.75	25.5	23.25	93.25
17	7.75	28	8.25	2.75	26.5	27	100.25
18	7.75	28	8.75	2.75	29.25	29	105.5
19	7.75	28.75	8.75	3.25	29.75	34.25	112.5
20	7.75	29.5	8.75	4	29.75	36	115.75
21	7.75	29.5	8.75	4	29.75	36.5	116.25
22	7.75	29.5	8.75	4	29.75	36.75	116.5
23	10.25	29.5	8.75	4	29.75	36.75	119
24	10.25	29.5	8.75	4	29.75	38.25	120.5

Table 1. Breakdown of Verification Effort over Time. This table shows the cumulative amount of time spent on formal verification up to the nth week of the project. Times are in hours, recorded in 15 minute increments. "Overhead" indicates time spent on general overhead, such as setting up computers or installing software. "Meetings" indicates time spent in meetings; the time was multiplied by the number of people present if the meeting was for the formal verification project. "Study" indicates time spent studying the design. "Model" indicates time spent deciding how to model the design for model checking. "Code" indicates time spent actually coding the model. "Run" indicates time spent waiting for verification jobs to run; it does not include the time taken by jobs left unattended. Most interesting verification traces and possible bugs occurred around the tenth week, as the basic model was completed and before lengthy runs were required. After the twentieth week, the project started winding down, with very large verification jobs left to run unsupervised on idle machines.

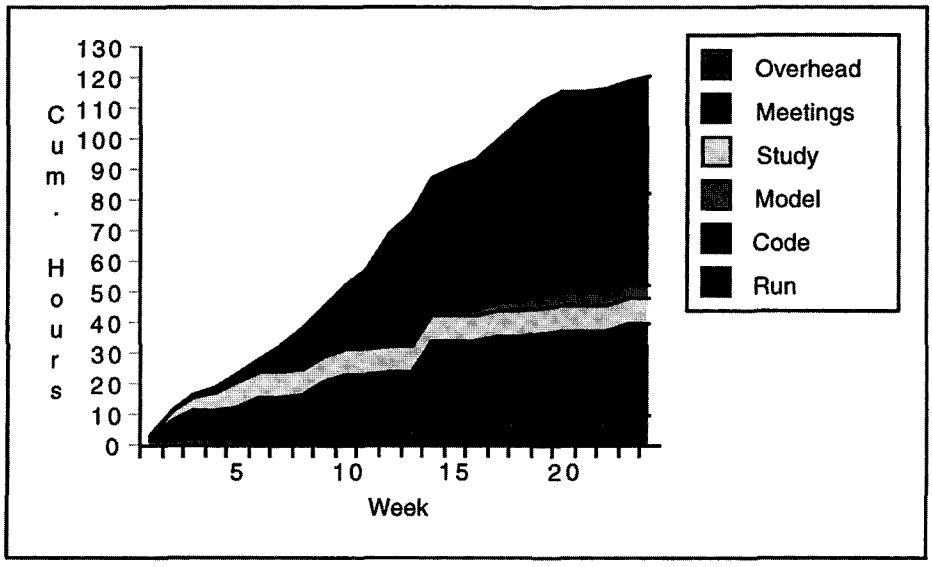


Fig. 5. Verification efforts

The main weakness to protocol-level formal verification is the lack of connection between the formal verification and the overall validation effort. Not finding bugs at the protocol level does not imply an error-free implementation. Nevertheless, finding any bug faster via formal verification than by other methods is an enormous benefit, making formal verification worthwhile if it's sufficiently easy or sufficiently likely to find difficult bugs. In the future, superior verification tools that allow verifying models much closer to the actual hardware, or new verification techniques that use the information gleaned from higher-level verification to help test lower-level implementations will greatly enhance the value of formal verification.

4 Future plan

Right now formal verification techniques can be applied to real design if verification experts are available. Typical designers have lots of interest in formal verification techniques, but unfortunately they cannot use them effectively, simply because it needs significant amount of human interaction with formal verification tools even in the case of model checking, i.e., we need to create simplified/abstracted models for verification manually.

The main problem is always so called "state explosion" problem. New research results such as, automatic abstraction/reduction, compositional/modular verification, are promising, but still need some more time for practical use.

On the other hand, it is effective to use formal verification techniques when simulating designs. It needs lots of efforts to manually create good simulation patterns. So one way is to use formal verification oriented techniques to generate effective simulation patters. This approach can be of practical values in the near future.

References

[CFFT96] Tom Chaney, J. Andrew Fingerhut, Margaret Flucke, and Jonathan Turner. Design of a gigabit ATM switching system. Technical Report WUCS-96-07, Computer Science Department, Washington University, St. Louis, Missouri, February 1996.

[CYF94] B. Chen, M. Yamazaki, and M. Fujita. Bug identification of a real chip design by symbolic model checking. In *Proceedings of the European Conference on Design Automation, the European Test Conference*, pages 132–136, Paris, France, February 1994. IEEE Computer Society.

[HFW97] A. Hu, M. Fujita, and C. Wilson. "formal verification of the hal s1 system cache coherence protocol". In *Proc. of International Conference on Computere Design*, Oct. 1997.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Pub., Boston, MA, 1993.

[MF85] F. Maruyama and M. Fujita. Hardware verification. *IEEE Computer*, (2):22–32, February 1985.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

Formal Methods in the Specification of the Emergency Closing System of the Eastern Scheldt Storm Surge Barrier

Meine van der Meulen[1] and Tim Clement[2]

[1] SIMTECH ENGINEERING BV, Rotterdam
[2] Adelard, London

Abstract. This paper describes the use of formal methods in the specification of the Emergency Closing System of the Storm Surge Barrier in the Eastern Scheldt (The Netherlands). Formal methods have proved to be very useful in obtaining an exact specification of the system and identifying possible errors.
Formal methods also pose problems. Much depends on the communication between the client and the expert and, since the client is not able to assess the work of the formal methods specialist, the trust between them.

1 Introduction

The Eastern Scheldt Storm Surge Barrier is one of the largest storm surge barriers in the world, and special because it is normally open and only closes in emergency conditions. The barrier stretches several kilometres with three subbarriers and two artificial islands. It protects the province of Zeeland from storm surges from the North Sea. It normally remains open to protect the environment in the Eastern Scheldt.

Since its construction in the eighties, technology has changed. One of the systems that needs upgrading is the Emergency Closing System (NSS). Normally the decision to close the barrier is taken by the operators, based on predicted water levels. However, if this is not done for some reason and actual water levels reach dangerous positions, the NSS will close the barrier automatically. The NSS may also reopen the barrier after the surge, in the absence of manual intervention.

Inputs to the NSS are provided by six water level sensors, three inside the barrier and three outside. Its outputs go to a starting automaton, which controls the opening and closing of the barrier.

The activities for the renewal of the NSS comprise:

1. Writing the requirements specification.
2. Writing a project plan.
3. Performing a project risk analysis.
4. Formally proving parts of the specification.

5. Identifying potential suppliers.

This paper describes work done on formally proving parts of the specification. The parts considered are the safety critical parts of the NSS.

2 Functionality of the NSS

The safety function of the NSS is:

- To trigger the starting automaton when the water levels exceed preset trigger levels and the outside water level is higher than the inside water level.

The following functions are related to the safety critical part of the NSS, but are not safety critical in themselves:

1. To open the barrier when the outside water level is lower than the inside water level again.
2. To allow the operator to block automatic opening by the NSS.
3. To allow the operator to change the trigger level for the inside water level.

The NSS also performs other functions, such as the presentation of water level information, but these are less safety critical.

3 Philosophy of the new specification

Since the eighties, much has happened in thinking about safety critical systems. The new NSS reflects these changes in philosophy.

First of all, the specification closely follows IEC61508, 'Functional safety of electrical/electronic/programmable electronic safety-related systems' [1], a standard that reflects the latest developments in this field. This standard not only considers the system itself, but also addresses design for assurance, the way to arrive at and use safety critical systems, and so on.

We have also made extensive use of the "keep it simple"- strategy. Unnecessary communication has been removed, as well as unnecessary commands. Also, we simplified the voting strategy on the sensors and the security approach. A second effect of the "keep it simple"-strategy is that we decided to implement the safety critical part of the NSS with hardwired logic. Hardwired logic for safety applications is readily available on the market, and it makes it unnecessary to reason about the safety of compilers, text editors and the like.

Most important of all in this context: we formally proved several safety properties defined in the specification. This is a necessary activity to justify claims for the higher Safety Integrity Levels in IEC61508.

4 Differences between the present and new NSS

The new philosophy leads to changes in the specification. These are most visible in the strategy of coping with sensors, the algorithm to open the barrier and the approach to security.

4.1 Coping with sensors

The present NSS uses an elaborate algorithm for coping with sensor information. This involves calculating an average value of the water levels, comparing values and disabling sensors when measured values differ too much from the average.

We considered this algorithm rather complex and preferred to simplify it. The new algorithm compares all measured levels with trigger levels separately, and then votes 2oo3 to determine whether a water level exceeds the emergency conditions.

A second major change is that the operator can no longer set the trigger levels to arbitrary levels within a range. It appeared that for the outside water level law sets the trigger level and for the inside water level only two trigger levels are being used. We decided to only implement these three trigger levels.

4.2 The open algorithm

The decision to use hardwired logic for the safety-related part of the NSS influenced the design of the open algorithm. The present solution is time discrete; the hardwired solution will be time continuous.

The present NSS can give an open command after the barrier closes, and the outside water level becomes lower than the inside water level. The operator can block the open command by typing in a command (VO).

The present approach has the following disadvantages:

1. The operator cannot revoke the VO-command.
2. There are no security precautions that someone else issues the VO-command, except for access control to the room where the system is installed.
3. The present system contains a slight error, that it issues the VO-command itself, but only as a consequence of a very rare sequence of events.

Therefore, we have changed the approach to the VO-command. It is now implemented using a key-operated switch. The operator can block the NSS from opening the barrier using this switch. He can undo his command as long as the closing conditions are present. He gets feedback on the status of the NSS (will it open the barrier or not) by a lamp.

4.3 Security

We also took another approach to security. In the present system the only security measure is control of access to the room containing the system. The new NSS uses key-operated switches for its two remaining commands. Here, it is important that these two commands are not safety-critical. The NSS will dependably close the barrier and open it again when the operator is not present or forgets his keys.

5 Safety argument

In [2], we find the following general structure for arguments about the safety or other properties of a system. The support for each claim comes from pieces of evidence, perhaps in the form of subclaims, which are justified in turn, combined by suitable inference rules.

For the barrier, the claim is that it meets the dependability constraints. These are sufficiently stringent that the system as a whole must continue to operate correctly even when some components have failed. The form of the argument adopted is that for any combination of failures of the components (including no failures), either

1. The system works correctly; the evidence for this can be provided by formal proof. or
2. The failure mode is sufficiently rare that all such failures do not exceed the allowable failure rate; evidence here is provided by failure rate data.

The choice of the partitioning between these possibilities is arbitrary, and largely based on engineering judgement. Typically, we might want to argue that all single failures are tolerated, and that multiple failures are sufficiently rare. (This of course requires some method of revealing latent faults.)

The formal proofs depend on:

1. The specification of the system functions, e.g. the conditions for the barrier to close.
2. The specification of the primitive components, including specifications of their failure modes.
3. Definitions of the system design: how the components are connected.

For each failure condition to be shown to be tolerated, we then prove that the design meets the specification given the behaviour of its components.

6 Controlling complexity in the arguments

There are too many components and too many failure modes to consider being able to deal with each independently: to control the work involved we must factor the proof. To do this, we exploited the modularity and symmetry of the design, and made use of loose specifications to describe a variety of failure modes in one definition.

To exploit modularity, we treat modules of the design as if they were components of the top level, giving specifications of their correct function and failure modes. We may then be able to show that some failures of modules are tolerated by the top-level design. These failures can be allowed to be produced by the module design when its components or submodules fail. This controls complexity because we are dealing with fewer components at each level. The number of failure specifications does not increase significantly because the size of the interface to each module is comparable with that of an individual component. Looseness of specification fits well with voter-based redundancy. Informally, a 2oo3 voter does not care about one input. Formally, we can specify that when

the module providing the input fails, we know nothing about the output. This covers all possible particular failure modes. One proof shows that the voter does not care, and so all failures of the input module are tolerated.

Symmetric design (which is typical of voter-based redundancy) also simplifies the proof obligations considerably because there are many similar proof cases. We can use one proof as the basis for writing another, preserving the strategy in our heads, or we can express the strategy formally (in the guise of a tactic) and use it on all cases. It may even be adequate to use the mathematicians approach of just saying similarly.

7 Mechanical proof support

We used HOL90 [3] to do definitions and proofs. One advantage of HOL for our purposes was its support for higher order functions. These allowed us to write module designs as functions from the specifications of the components to the behaviour of the module, in effect capturing just the module wiring. It was then easy to plug in a mixture of failed and working component specifications to define a module with a particular failure set: another route to controlling the complexity of the proof work.

Proofs in HOL are described using a language of tactic applications, which is well above the primitive inference level of the logic but rather below the level at which people describe proofs. We can have high confidence in the correctness of HOL proofs, because:

- Its approach is one of giving definitions, not introducing axioms, so the theory is as sound as the underlying logic (and this logic is well understood and small).
- Proofs must be valid if the HOL90 implementation is correct. The theoretical justification for the claim of correctness is that the proofs are protected by the strong type system of ML, the implementation language of HOL. The practical justification is that the tool has extensive field experience.

The proofs established the expected correctness of the 'close' function, including a sample of failure modes, once one slip where two signals were accidentally swapped had been corrected. The sample of failure modes could easily be extended.

Contemplating the proof of the 'open' function helped to refine the requirements, and identified faults in an initial design of the new NSS. However, when the proofs of the final design were carried out, there remained aspects that were not completely modelled and differences between the model and the actual implementation that limit the strength of the claims of reliability that we can make from them. These stemmed in part from a difference between the natural way of expressing the design at a circuit level and the simplest approach for reasoning about it in HOL. We are continuing to try to close these gaps. The mere existence of a proof does not guarantee that the system is correct. To have confidence in the system, it is necessary to review what has been proved, looking in particular for:

1. Sins of commission: the definitions may not capture the requirements and the design. These are easier to spot than
2. Sins of omission. Formalisation (indeed, any form of description) inevitably involves abstraction. The system may fail in the unaddressed areas of reality. One particular area of concern with the barrier is the length of time it takes to close and open, which has not been modelled.

To achieve an effective review, the client (who has the understanding of the problem domain) has to be involved. Using HOL, there is a considerable barrier of notation to overcome even to communicate the definitions of the design and its properties, which must be reviewed. In principle, we can take the proofs themselves on trust given our confidence that HOL constructs only valid proofs. In this project, this required not only that Simtech should have confidence in Adelard, but that they should be able to convey that confidence to their client. For this reason, we did put some effort into communicating how the proofs worked as well. This raised the barrier of notation even higher: the HOL tactic language is large and HOL proofs mix details of symbol pushing with the larger strategic decisions that proofs communicated between people concentrate on. One rule of thumb for extracting the latter is to look at points in the proof where the goal splits: this is characteristic of interesting steps such as case analysis, introduction of lemmas, and use of induction. This was not, however, enough to solve the problem of communication completely within the timespan of the project.

8 Conclusion

Formal methods proved to be very useful in the specification of the Emergency Closing System of the Storm Surge Barrier in the Eastern Scheldt. They helped to specify exactly what was intended and to capture the design, and hence to identify possible errors.

There also where some complications using formal methods in an industrial setting. Communication is difficult. To specify the system formally, the client has to be very precise about the functionality. This is good only if it does not impose a formalism for the specification that does not fit with from his normal pattern of description. The client also has the problem of assessing the work of the formal methods specialist. Even taking on trust those areas, which need not be of concern, can be difficult, since the client has to trust the specialists claim that this is the case.

References

1. International Electrotechnical Commission, IEC61508, Functional safety of electrical/electronic/programmable electronic safety-related systems.
2. Adelard Safety Case Development Manual, Adelard, 1998. ISBN 0 9533771 0 5.
3. Introduction to HOL: A theorem Proving Assistant for Higher Order Logic. M.J.C.Gordon and T.F.Melham, Cambridge University Press, 1993.

The New Topicality of Using Formal Models of Security Policy within the Security Engineering Process

Frank Koob, Markus Ullmann, Stefan Wittmann

Bundesamt für Sicherheit in der Informationstechnik
Godesberger Allee 183
D-53133 Bonn, Germany
ullmann/koob/wittmann@bsi.de

Abstract. This paper is focussed on the notion of a *Formal Model of Security Policy (FMSP)*. This kind of model is essential when reasoning about the security of Information Technology devices like a specific IT-product or IT-system. Without an unambiguous definition of what security means, it is impossible to say whether a product is really secure.

1 Introduction

The growing internetworking and spectacular security weaknesses in IT-systems generate growing need in third party evaluation. Actual criteria require the usage of formal methods for the development and evaluation of security products at specific levels of assurance. Current IT evaluation criteria like the Information Technology Security Evaluation Criteria (ITSEC) [ITSEC] or the Common Criteria (CC) [CC] take this into account and require an FMSP at assurance level E4 (ITSEC) or EAL5 (CC), respectively. This is one reason that the usage of formal methods becomes more popular in the security community today.

In general a first euphoria in using formal techniques for the development of secure systems came up at the end of the 70's. There was a big horizon of expectations in the usage of formal methods especially in the US for the development of high assured IT-products in this timeframe. But these expectations did not come true. A lot of reasons are responsible for that.

Since that time a large amount of formal methods research was done up to now. This leads to more usable, scalable formal methodologies and accordingly integrated provers, see as an example [KUW95]. In the meantime experts understood their structured use for the development of highly dependable products very well. But the usage of this technique is changing over from *program verification* to the most critical early phases of product development. FMSPs are only one example for this tendency.

This paper is structured as follows. In general there is a lack of understanding of what FMSPs are usable for. They are regarded as something exotic by the community being not familiar with formal techniques. For that reason we firstly describe in chapter 2 that the usage of models is in principal a usual procedure

in the technical area and science. In the next chapter 3 we state, that there is no exact definition of what security really means. But if it is necessary to reason about the security of an IT-product a well definition of the meaning of security in that context is essential. In the chapter 4 we describe the notion of FMSPs as they are understood in the context of ITSEC and CC. In the final chapter 5 a conclusion is given.

2 Mathematical Models

In science, a model is a mathematical representation of some behaviour. They are often called "laws" but they are not mandatory in the sense of a law at all. For example in electronics models are just relationships between measurable quantities that generally are useful in predicting behaviour in a way that allows us to design and build products of various kinds. In general a *model* is a reflection of the reality with special emphasis on important properties. Irrelevant aspects are out of consideration. A model is in this understanding a specific abstraction of the most important aspects. Typically models have three characteristic properties:

- *descriptive:* the model describes interesting and important situations of the reality
- *explanatory:* models support the explanation and the deduction of noticed phenomena
- *predictive:* models support the verifiable prediction of phenomena

By being predictive, the model helps us to understand what will happen in a situation that we have not previously observed, or perhaps can not observe.

The predictive nature also helps to validate the model. When models predict behaviours that are demonstrable false, one may conclude that the model is incorrect or is being applied to problems to which it is not suitable. For example, Newton's laws do not apply to particles moving at speeds near that of light.

Models are developed through a process of abstraction. In reality, concrete things have imprecise, often vague, semantically ambiguous descriptions. Through a process of abstraction we develop a mathematical representation, a model that has a precise, unambiguous description, but is not something that exists in the real world.

But we always must have in mind that this process of abstraction might turn to specific dangers. Normally the concentration on the important principles within a model shows more transparency then the reality. But the problem is to find the right level of abstraction without loosing the reference to the reality.

One example of a very simple model is *Ohm's Law*.

$$U = RI \qquad (U : \textit{voltage}, \quad R : \textit{resistance}, \quad I : \textit{power})$$

This equation is not absolutely exact in all situations. R is only approximately a constant. The resistance of a material depends on temperature etc. . Strictly

speaking this equation is an idealisation of the real behaviour. Models represent real ambiguous descriptions by unreal (or less real) things that have a precise description.

Besides the above mentioned notion of a model there is still another interpretation of the model in mathematics. In mathematical logic a model is a structure which fulfils an existing set of axioms. The existence of such a structure guarantees that this set of axioms is without contradiction.

Normally formal models of security policies are formalised in mathematical logic. But they are models in the sense of reflection the reality. They will define relationships between certain sets that may help us design and build more trustworthy IT-products and systems.

3 Definition of Security

In the context of the ITSEC, security means:

- *confidentiality:* information is only disclosed to those users who have authorised access;
- *integrity:* information is modified only by those users who have the right to do so;
- *availability:* information and other IT resources can be accessed by authorised users when needed.

This definition gives the impression that security is an objective notion. But it is not. The following description forms a clearer picture of the notion of security:

> *Security is the property of a system, which is characterized by the fact that threats regarded as being important, directed against the protect-worth goods can be precluded by special countermeasures to a degree that the remaining risk is accepted.*

This definition shows that security is in principal a subjective notion. This is the reason that there is no formal definition of security at all that takes all intuitive ideas of security into account. For this reason FMSPs are so important because they define unambigously the specific kind of security established in a specific IT-product or IT-system. Out of the large number of existing FMSPs some are mentioned below.

Bell-La-Padula Model: It describes a mandatory access control policy that is focused on the nondisclosure of confidential information. [Bel73]

Biba Model: This model is concerned only with preventing unauthorised modification of information (integrity). This model has some analogy with the Bell-La-Padula model. [Bib75]

Signature Model: It describes the nonrepudiation and authenticity of digital signatures. In contrast to the Bell-La-Padula model and the Biba model the main point is not the description of a subject-object-relation but the description of sequences of actions concerning the generation and verification of digital signatures. [BSI98]

Noninterference Model: An intuitive definition is found in [Rus92]:

The idea of noninterference is really rather simple: a security domain 'u' is noninterfering with domain 'v' if no action performed by 'u' can influence subsequent outputs seen by 'v'.

Noninterference models are very abstract and they give no guidance in implementing such a model. They were used to formalise channel-control security policies.

The idea of FMSPs is to formalise the intention of security in a very precise but abstract manner and to do a lot of proofs, as we describe in the next chapter. For example we can prove that an abstract state machine with transition functions satisfies certain constraints and preserves other properties as invariant under transition. Nothing has been proved about the real product. Only to the extent that the product is a realisation of the FMSP do we have any assurance. The question whether the FMSP is relevant to the real product is not a question of mathematics. The question is, does the model represent the real product on an abstract level? Only if the components of the model correspond to real things in the IT-product then we have confidence that the security of the real IT-product follows from the security of the model. That is nothing new and valid for all kinds of models.

Typically a lot of assumptions are made within the model. On the other side a lot of threats have influence in the security and effectiveness of real security mechanism. This means the evaluator has to check during the evaluation process whether this attacks influence the assumptions of the model.

An example may clarify this a little bit. In the Signature Model there are a lot of assumptions. In the model a *hash-function* is not described in an algorithmic style, only the property of this function is described:

$$hash(hm1, dd1) = hash(hm2, dd2)$$
$$\rightarrow \quad dd1 = dd2 \; AND \; hm1 = hm2$$

This means that given a hash value it is impossible to find two different documents that have the same hash value. This is certainly an idealisation of real hash-functions. But it is known for security reasons of digital signatures that hash functions must fulfill this property as good as possible.

The IT-security evaluation criteria as described in the next chapter consider the problems mentioned in this chapter. There are strong links between the FMSP and other it-product concerning evaluation documents and the evaluation process as such.

4 Formal Model of Security Policy according to ITSEC and CC

4.1 Relation to the Evaluation Process

Each evaluation based on the ITSEC starts with a document called *security target*. This document written in natural language describes the target of evaluation (TOE) on an abstract level. Normally it starts with a description of the

product rationale. This gives an overview of the functionality of this product and identifies the TOE very exact. Each IT product will have its own specific requirements for maintenance of confidentiality, integrity and availability. This requirements are called *security objectives* of the TOE within the security target. On the other hand we have assumed threats which can have an effect on the security of the product. To repulse this threats (attacks) the product needs a specific security functionality and according security mechanisms. Furthermore a product has an assumed environment which is described also in the security target. Some parts of the security target are very important for the construction of the FMSP:

- *assets of the IT-product*
- *security objectives of the IT-product*
- *assumed threats*
- *assumption about the environment of the IT-product*
- *security functions (mechanism) of the IT-product*

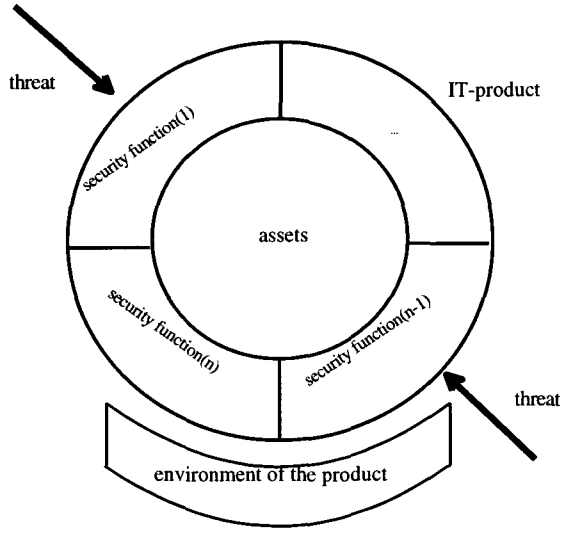


Fig. 1. assets, assumed threats and security functions of the IT-product

The idea is, to formalise that in a specific kind described in the final chapter.

For clarification reasons we give a very small example. This is done only to give an impression of the close connection between the security target and the FMSP. The focus is on a digital signature application which is processed within an IT-product.

Here we do not describe the *rationale* of a product with the concerning description of the IT-product as such. That is outside the scope of this paper.

The TOE is connected to some system in order to use the cardholders secret key stored in an protected environment of the TOE. The digital signature operation and the verification of received signed documents itself is performed within the TOE.

For this fictive IT-product only the security objectives and one property of a security function are described:

Security Objectives: If a received signed document is validly checkable based on well defined security actions, then the document is authentic and nonrepudiatability holds. This principal security objective is formalised in the following axiom:

$$
\begin{array}{l}
nonrepudiatable1(w) \\
\quad \leftrightarrow \quad ALL\ p, d, al : \\
\quad\ checked_ti(w, p, d) \\
\quad\quad \rightarrow \quad EX\ al1 : \\
\quad\quad\ check_listp_ti(all, p, d) \\
\quad\quad\quad AND\ checked_ti(\\
\quad\quad\quad\ interpret(reset_ti(\\
\quad\quad\quad\quad interpret(w, al), p, d), al1, p, d))
\end{array}
$$

Here nonrepudiation is expressed based on the reproduction of the signature validation result. A signed document with timestamp was validly checked (***checked_ti***). After arbitrary actions may have taken place ($interpret(w, al)$) the signature validation result is reset ($reset_ti$) and if the check of the signature holds again nonrepudiation is valid.

Security Functions: A signature can only be generated with the intention and awareness of the cardholder.

We can formalise this properties by definition of a predicate *agree*.

$$
\begin{array}{l}
agrees(interpret_h(w, al, n), p, d) \\
\quad \leftrightarrow \quad agrees(w, p, d) \\
\quad\ OR\ EX\ dd, prk : \\
\quad\quad d = present(the_s m, dd) \\
\quad\quad\quad AND\ ExvalidAuth(w, al, n, p, the_sm, dd, prk)
\end{array}
$$

The example is out of [BSI98]. Details and broader explanation are found there.

We have made the experience, that the consequent analysis of the informal security target as the beginning of a following formalisation process shows some insufficiencies of the informal description in the target. These are:

- incompletenesses
- inconsistencies
- ambiguities.

This global results shows the importance of formal security policy models for the evaluation of IT security products and systems.

4.2 Formal Mathematical Notations

The formal notation must be based upon well-established mathematical concepts. These are for example abstract state machines or abstract data types used in formal specification languages or constructs from logics like predicate logic, temporal logic and set theory.

4.3 Components of Formal Models of Security Policies

In general a FMSP consists of the following parts:

- abstract description of the important principles of security written in a formal mathematical notation
- formalised assumptions of the product environment
- definition and proof of the global security objectives
- proof of the internal consistency of the security principles

A common mathematical description technique to formalise the security behaviour of an IT-system are abstract state machines. The Bell-LaPadula model for instance was the first approach where this technique was used. When abstract state machines are used, the first thing to do is to define entities which represent the security relevant state variables. Typically these are subjects, objects, attributes (e. g. access rights) etc. . To define deterministic behaviour an initial instantiation of the state variables is necessary. This is called the initial state. The possible operations of the abstract state machines are expressed with transition functions. Here typically the rules of operation and request for access are expressed which we call security functionality. As a rule, FMSPs are defined in an abstract manner. This means for instance that the defined data structures are not real in the sense, that they are implementary directly with an programming language like Java or C. On the other hand sometimes statements on specific behaviours or properties of used mechanism are needed, see the example with the properties of hash-functions as described in chapter 3. As an assumption it was demanded that this function is injective. This means that if you have calculated a hash value for a specific document it is impossible to find a second document with the same hash value. We know that this assumption is an idealisation of real hash functions. In a real evaluation the evaluator has to validate, whether the chosen real hash algorithm in the IT-product satisfies this assumption as good as possible. One part of the FMSP is the definition and the proof of the security objectives. With the abstract state machine we can do that with a description of secure states (properties derived from the global security objectives). The next step is the proof that each reachable state of the product is a secure state (this means that the global security objectives are invariants of the product). The last thing to do is to proof the internal consistency of the security principles. This can be done by showing the existence of an implementation which has to be a correct refinement of the abstract model. For this you can choose any implementation (no memory limitations, efficiency is not required,

etc.). This internal consistency is necessary to guarantee that the proof of the global security properties are not based on the implication:

$$FALSE \rightarrow TRUE$$

Today there are quite a variety of computer based semi-automatic prover systems that efficiently reduce the proof effort. Examples are the *Verification Support Environment(VSE)* [KUW95], [Hut96] the B-Tool, KIV or pvs. Additionally it is very helpful for the evaluator if this kind of tools are used. Then the possibility of replaying the proofs exists. This highly reduces the effort of examine the proofs for the evaluator.

5 Conclusion

The increasing number of Formal Models of Security Policies required by security evaluation criteria not only enforces the use of Formal Methods but also makes the benefits of this methodology for the software development process quite evident. Software engineers and developers are forced to define security objectives uniquely related to the system or product to be realised in order to meet special security threats. The formal modelling process confirms the adherence to invariants and detects contradictions and inconsistencies.

What is still needed for better understanding of FMSPs is the existence of good examples which shows the collaboration of the different documents like security target, FMSP and so one. Some work in the field of digital signature components along the german signature legislative [GSL97] is in preparation.

References

[Bel73] Bell, D.E. and L.J. LaPadula, Secure Computer Systems: A Mathematical Model, ESD-TR-73-278, Vol.II, November 1973, The MITRE Corporation, Bedford, MA: HQ Electronic Systems Division, Hanscom AFB, MA.

[Bib75] K. J. Biba, Integrity Considerations for Secure Computer Systems, The Mitre Corporation, Bedford, MA, MTR-3153

[BSI98] Bundesamt für Sicherheit in der Informationstechnik, Formales Sicherheitsmodell zur Erzeugung und Prüfung digitaler Signaturen mit Hilfe von VSE, BSI report, 1998

[CC] Common Criteria for Information Technology Security Evaluation (CC), Version 2.0, http://csrc.ncsl.nist.gov/nistpubs/cc/

[Gov93] Ronald A. Gove, The Theory and Construction of Formal Security Policy Models, Tutorial 9th Annual Computer Security Application Conference, Orlando, 1993

[GSL97] Gesetz zur Regelung der Rahmenbedingungen für Informations- und Kommunikationsdienste (Informations- und Kommunikationsdienstegesetz - IuKDG) (in Kraft getr. am 1.8.97) Artikel 3 Gesetz zur digitalen Signatur (Signaturgesetz - SigG)

[Hut96] D. Hutter, B. Langenstein, C. Sengler, J. Siekmann, W. Stephan, A. Wolpers, Verification Support Environment, Proceedings Formal Method Europe-96, M.C. Gaudel and J. Woodcock (Eds.), Springer Verlag, LNCS 1051, 1996, 268-286

[ITSEC] Information Technology Security Evaluation Criteria, Office for Official Publications of the European Communities, Luxembourg 1991.

[Kes93] Volker Kessler, Der Sinn von Sicherheitsmodellen, KES 93/6, 1993

[KUW95] Frank Koob, Markus Ullmann, Stefan Wittmann, The Formal VSE Develoment Method - a Way to Engineer High-Assurance Software Systems, Proceedings of the 11th Computer Security Application Conference, pages 196 - 204, IEEE Computer Society Press, New Orleans, 1995

[Lam71] Lampsons, B.W.,Protection. Operationg Systems Review, Januar 1974. 8(1):p. 18-24. Originally published in Proceedings of the Fifth Princeton Conference on Information Sciences and Systems, March 1971

[Lan81] Landwehr, C.E. Formal Models for Computer Security. Communications of the ACM, October 1973. !6(19):p. 613 - 615

[Rus92] J. Rushby, Noninterference, Transitivity and Channel-Control Security Policies, Technical Report CSL-92-02, SRI International, 1992, available at http://www.csl.sri.com/rusby/reports/csl-92-2.dvi.Z

[Ste92] Angelika Steinacker, Sicherheitsmodell für informationstechnische Systeme - Leitlinien für die Entwicklung, Datenschutz und Datensicherung 1/92, S. 17 - 21, 1992

Towards Comprehensive Tool Support for Abstract State Machines: The ASM Workbench Tool Environment and Architecture

Giuseppe Del Castillo
giusp@uni-paderborn.de

Heinz Nixdorf Institut, Universität-GH Paderborn
Fürstenallee 11, 33102 Paderborn, Germany

Abstract As a framework for the systematic development of tools supporting the development and analysis of Abstract State Machine (ASM) specifications, we developed the *ASM Workbench*. This paper describes the basic concepts of its architecture and the existing components.

1 Introduction and Motivation

There is general agreement on the fact that appropriate tool support is needed in order to gain practical advantages from the application of formal methods to concrete specification and modelling tasks. Useful tools include:

- syntax-directed and graphical editors;
- static checkers (e.g., syntax and type checkers);
- prototyping/simulation/animation tools and code generators;
- verification tools (model checkers, proof checkers, theorem provers).

This consideration obviously applies also to the method of Abstract State Machines (ASMs) [10]. In particular, potentially useful ASM tools can be roughly subdivided into the following two classes:

1. tools supporting a user in the process of developing ASM specifications, e.g., editors, static analysers, interpreters, debuggers: they assist the specification writer in formalizing informal descriptions of systems or requirements and improving such formalizations as the result of an iterative process, involving experimentation as an important means of specification validation;
2. tools transforming ASM specifications into other languages in order to allow the processing of these specifications by other tools, e.g. code generators transforming ASM specifications into programs or translators mapping them into the language of a particular logic: tools of this kind enable the use of state-of-the-art compilers and verification tools to produce efficient executable code or to verify properties of the specifications, respectively.

Note that, while the former tools (*interactive development tools*) are helpful during the formalization phase, the latter (*transformation tools*) come into play after it, when the ASM models formalizing the problem at hand are completed.

The current state of the art of tool support for Abstract State Machines—despite the efforts of many people and the improvements achieved during the last years (see Sect. 4 for an account of the existing ASM tools)—is still far from being completely satisfactory.

On one hand, there are several ASM simulators (mostly interpreter-based), which usually—besides the possibility of executing ASM specifications—do not provide any other interesting feature.[1] With the exception of the Aslan tool [1], which includes a debugging feature, these simulators provide very restricted possibilities of user interaction, therefore they are not very convenient as interactive development tools. Moreover, many of them rely on the implementation language in order to define certain parts of the specifications, such as static function definitions: this forces the user to write programs in this language to define particular aspects of a specification (at a level of abstraction which may be too low, e.g. C code) and presupposes the availability of a development environment for the implementation language[2].

On the other hand, there are essentially no trasformation tools. Although, for instance, some successful approaches to mechanical verification of properties of ASM specifications [20–22] by theorem provers (KIV [19] and PVS [18]) or model checkers (SMV [15]) employ transformations of ASMs that are quite general and thus subject to be automated, these transformations have not been implemented, probably due to the relatively high amount of implementation work needed (for the case studies considered in the papers the transformations have been applied by hand). However, the implementation effort could be drastically reduced if an appropriate framework/library for the development of ASM tools were available.

In response to the situation described above, we developed the *ASM Workbench*, a tool environment intended to support both the development process and transformations of ASMs, based on a open architecture providing basic functionalities and easily extensible.

In this paper we report about the ASM Workbench tool environment and its architecture. In Sect. 2 the main features of its source language ASM-SL, which extends the basic language of ASMs defined in [10], are introduced. In Sect. 3 the architecture of the ASM Workbench—with particular emphasis on its modular structure and the possibilities of extension—is outlined, and the tool components already developed are briefly described. Sect. 4 gives an overview of related work (existing ASM tools). Finally, we make some concluding remarks.

---

[1] A notable exception is the Gem-Mex tool [2], which includes a graphical editor for control/data-flow diagrams and documentation generation features: however, the Gem-Mex tool supports the special ASM-based methodology of *Montages* [2] for specifying programming languages, not ASMs in their general form.

[2] While this is usually not a problem for standard public domain tools like the GNU C/C++ compiler ot Tcl/Tk, it may become one if more exotic languages and/or commercial compiler implementations are needed.

2 The ASM-SL Notation

The source language for the ASM Workbench is the ASM-SL notation[3] [4], which extends the basic language of ASMs defined in [10] by a few additional constructs, addressing pragmatical issues which arise in applications. In particular, the extensions include:

1. a simple but flexible *type system*[4];
2. constructs to define the states (i.e., the algebras representing the *data model* underlying the behavioural model specified by ASM rules);
3. mechanisms to define *interfaces* to the environment, as needed for modelling open systems.

The language extensions have been realized by integrating into ASMs established concepts from other languages and methodologies, while trying to keep the language *concise*, *understandable*, and *executable*.

In this section, after discussing the language extensions, we shortly describe the structure of ASM-SL specifications and conclude with a simple example which shows some of the peculiar features of ASM-SL.

2.1 Type System

ASMs, as defined in [10], are untyped, but pragmatic considerations motivate the introduction of a type system. In particular, types provide a considerable help in clarifying and exposing the structure of the data model; moreover, a type-checker may detect trivial errors and inconsistencies very early, before undertaking any simulation or verification.

The type system of ASM-SL is an adaptation of the well-known type system of Standard ML [17], based on *parametric polymorphism*. This type system seems to be a reasonable compromise between simplicity and flexibility and has the advantage of being quite well known: additionally, the possibility of performing type inference allows for concise specifications, as most type declarations can then be omitted.

2.2 Definition of the Data Model

The issue of specifying the static part of ASM specifications, i.e. the data structures underlying ASM computations (called *data model* for short in the rest of this paper) is not considered in [10]. In fact, one could use for this purpose any of the existing approaches (e.g., algebraic or model-based specification); the so specified data model can then be easily combined with a *behavioural model* of the system specified by ASM rules, thanks to the abstraction of states as algebras.

The solution adopted in ASM-SL consists in providing:

[3] ASM-SL stands for ASM-based Specification Language.

[4] Of course, there are also good reasons for keeping ASMs untyped. Note, however, that the type system is orthogonal to the other extensions, which could in principle be interpreted as untyped as well.

- a set of *predefined elementary types* (booleans, integers, floating-point numbers, strings) and *predefined polymorphic types* (tuples, lists, finite sets, finite maps) together with a construct for defining arbitrary *freely generated types*, as a means of defining data types;
- a set of additional *language constructs* borrowed from functional programming and model-based specification[5] (recursive and mutually recursive function definitions, pattern matching, set-theoretic notation), as a means of defining operations over the data types (*static functions* in ASM terminology) and the initial interpretation of dynamic functions (specification of the initial state).

This approach, though less abstract than others (e.g., algebraic specifications), allows nevertheless to model a wide range of applications[6], is quite intuitive, and has the advantage that *executable models* are obtained by constructions. The use of familiar structures and notations from discrete mathematics and computer science ensures that ASM-SL specifications can be easily understood and translated into other languages (programming languages or logics).[7]

2.3 Interfaces to the Environment

As ASMs are often used to model embedded, reactive and, in general, open systems, the issue of *interfaces* between the ASM model of a system and (a model of) the environment into which the system is embedded is a crucial one.

ASM-SL supports the standard ASM abstraction mechanism of *monitored functions* (also known as *external functions*) as a means of defining such interfaces. Declarations of monitored functions may come with *constraints*, which define restrictions on their possible interpretations.

Currently, ASM-SL supports two kinds of constraints for monitored functions, namely *type constraints* and *finiteness constraints*. The meaning of type constraints is obvious. Finiteness constraints define, for each point of a function's domain, a finite set of values which the function is allowed to assume on that point: finiteness constraints allow to consider the evaluation of external functions in a given state as a kind of non-deterministic choice among a finite set of alternative values, thus improving the possibilities of computer-aided analysis of ASM specifications, e.g. by model-checking.

In order to model the environment even more precisely and to further restrict the set of possible runs of an ASM, the possibility of specifying *temporal*

[5] The choice of constructs and concrete syntax adopted in ASM-SL is chiefly inspired by Standard ML [17] and VDM [13].

[6] Of course, other approaches based on specialized (application domain-specific) data models may be more convenient for particular applications. A significant example is given by the *Montages* technique for specifying programming languages and the related tool Gem-Mex [2], which are based on a specialized version of ASMs, where the underlying data model consists essentially of control-/data-flow graphs.

[7] Most theorem provers come with predefined theories and most modern programming languages with constructs or generic class libraries to deal with the mentioned structures.

constraints on the behaviour of monitored functions (e.g., as formulae of linear temporal logic) should be considered for future extensions of ASM-SL.

How monitored functions are to be treated in practice is left open and in fact depends on the purpose of the tool. For instance, the ASM simulator which is part the ASM Workbench uses *oracles* as a quite general mechanism to deal with monitored functions (see Sect. 3.2, "Simulator") and uses the constraints to perform run-time checks on the admissibility of the function values provided by the oracles. A transformation tool (currently being developed) to translate ASMs into the language of the SMV model-checker [15] exploits the finiteness constraints in order to derive the (finite) ranges of the generated SMV state variables (which SMV then uses to make an exhaustive exploration of the space of behaviours, in order to check temporal properties).

2.4 Structure of ASM-SL Specifications

ASM-SL specifications are sequences of definitions of types, functions, and named transition rules (also called *rule macros*). As in other languages designed for interactive use (e.g., Lisp, Prolog, ML), the definitions are processed sequentially and stored into a definition *database*. Subsequent definitions may contain names of types, functions and rule macros which are already in the database (*definition before use* requirement).[8] Similarly, at a given moment the tool is able to type-check or evaluate any term containing only defined function names, any transition rule containing only defined rule macros, and so on.

A distinguished transition rule may then be declared as the *program* (see [10]): this rule specifies the transition transforming each state into its successor state (i.e., the computation step). Multi-agent ASMs can be specified by declaring the function *Self* (determining which agent makes a move at a given stage) as an external function ranging over a finite set of agents: this corresponds to an *interleaving* semantics for multi-agent ASMs (*sequential runs* in [10]).

2.5 An Example

In order to give the reader a flavour of ASM-SL, we present an example which is quite trivial, but is easy to understand and contains many of the additional constructs discussed in this section (freely generated types, function definitions, finite sets, finiteness contraints).

Table 1 shows an ASM that simulates a non-deterministic finite automaton $(\Sigma, Q, q_0, F, \delta)$, provided that types *SYMBOL* and *STATE* for Σ and Q, and functions *initial_state* : *STATE*, *FINAL_STATES* : *SET(STATE)*, and *delta* : $STATE \times SYMBOL \rightarrow SET(STATE)$, for q_0, F, and δ, respectively, are defined[9]: note that a step of the simulating ASM corresponds to a step of the simulated NFA. Table 2 contains an example of the static NFA specification.

[8] As an exception, mutually recursive functions and types can be defined by means of a special form for *simultaneous definitions*.

[9] The type $SET(\alpha)$ is in ASM-SL the polymorphic type of finite sets over α, thus $SET(STATE)$ is the type of finite sets of states.

```
dynamic function running       :BOOL   initially true
dynamic function accepted      :BOOL   initially false
dynamic function current_state :STATE  initially initial_state

external function input_symbol :SYMBOL

external function next_state :STATE * SYMBOL -> STATE
  with next_state (q_, x) in delta (q_, x)

transition NFA_step ==
  if running
  then if input_symbol = END_OF_STRING
       then accepted := member (current_state, FINAL_STATES)
            running  := false
       elseif delta (current_state, input_symbol) = {}
       then running := false
       else current_state := next_state (current_state, input_symbol)
       endif
  endif
```

Table1. ASM-SL specification of the ASM simulating NFAs

3 The ASM Workbench

As mentioned in the introduction, the ASM Workbench (ASM-WB for short) is not only intended as a collection of tools supporting the ASM method, but also as an open architecture to be used as a framework for the systematic development of further ASM tools.

The main distinguishing features of the ASM-WB architecture are its *kernel* (a set of program modules implemented in the functional language Standard ML) and a set of *exchange formats* (textual representations of the kernel's data structures, to be used for importing and exporting ASM-related information). The existing components (tools) of the ASM Workbench—implemented either as part of the kernel or as additional modules—include a *type-checker*, an interpreter-based *simulator*, and a graphical user interface (*browser/debugger*). Other modules are under development, in particular an interface to the model-checker SMV and a code generator for the Java Virtual Machine (see Sect. 3.2, "Ongoing Work").

In this section, after a description of the ASM Workbench architecture and of the existing ASM-WB tool components, we discuss different possibilities of using, extending, and modifying the ASM Workbench.

3.1 Architecture of the ASM Workbench

The Kernel The essential functionalities of the ASM Workbench are contained in its *kernel*, which consists of:

```
external function n :INT

freetype SYMBOL == { a, b, END_OF_STRING }
freetype STATE  == { q : INT }
static function initial_state == q(0)
static function FINAL_STATES  == { q(n) }

static function delta (q(i), x) ==
  if x = END_OF_STRING then {}
  elseif i = 0 and x = a then { q(0), q(1) }
  elseif i = 0 and x = b then { q(0) }
  elseif i > 0 and i < n then { q(i+1) }
  else {}
  endif
```

Table2. Specification of the NFA recognizing $(a|b)^* a (a|b)^{n-1}$

- a collection of *data structures* representing *syntactic objects* as well as *semantic objects* of ASM specifications (collectively denoted as *ASM objects*). Syntactic objects are, for instance, terms, transition rules, and signatures, while semantic objects include *values* (i.e., the elements of the ASM *superuniverse* or *base set*), states, update sets, runs, etc.;
- a collection of *functions* to process the ASM objects, to be used as building blocks for the construction of more complex functions up to complete tool components. The kernel provides, for instance, functions to type-check a term or rule, to evaluate a term to a value or a transition rule to an update set, to fire an update set in a given state, and so on.

The kernel is implemented as a set of Standard ML *modules*, each module corresponding to a relevant data structure (e.g., abstract syntax trees, signatures) or functionality (e.g., type-checker, evaluator). There is also a collection of auxiliary generic modules, which can be reused for the development of other tools: it includes modules for importing and exporting ASM objects (see "Exchange Formats" below), a module for accessing the ASM-SL definition database, and a module implementing structural-inductive transformations (morphisms) over the ASM abstract syntax as higher-order functions.

Thanks to a feature of the Standard ML compiler, it is possible to export an executable image of the ML compiler itself (usually called `sml-asm`) containing the precompiled and preloaded kernel. In this way `sml-asm`, besides constituting a first—not very friendly—user interface for the ASM Workbench, provides immediate access to the data structures and functions of the ASM-WB kernel, which can be directly used for the development of further—tightly coupled—tool modules. (Table 3 shows an example session with `sml-asm`).

Exchange Formats Most data structures of the kernel come with corresponding textual representations (*exchange formats*), to allow importing and exporting

```
- ASM.load_file' "nfa1.asm"; ASM.load_file' "nfa_behaviour.asm";
val it = ["n","SYMBOL","STATE","initial_state","FINAL_STATES","delta"]
  : string list
val it = ["running","accepted","current_state","input_symbol",...]
  : string list
- ASM.eval_term' "delta";
[interactive definition]:1: [1(1)-1(6)]:
  type check error -- function delta : (STATE * SYMBOL) -> SET (STATE)
                      has argument of type ()
uncaught exception Error
- ASM.eval_term' "delta (q(0), a)";
val it = FSET (ref (FSet (T #),fn)) : ASM_Domains.VALUE
- ASM.show_value it;
val it = "{ q (0), q (1) }" : string
```

Table3. Example Session with `sml-asm`

ASM-related information: this enables loosely coupled tool integration. For instance, one could use the ASM-WB parser and type checker as the front-end for a transformation tool. The transformation tool would take as input the annotated abstract syntax trees exported by the type-checker (see below): the transformation tool does not need to care about the ASM-SL concrete syntax and can rely on the fact that its input represents a well-formed and type-correct ASM (where all the "syntactic sugar" has been eliminated). Such an approach may contribute to a considerable reduction of the overall implementation effort, while not requiring any knowledge of the internal workings of the ASM Workbench.

3.2 Components of the ASM Workbench

Type Checker A type checker, implementing the type system discussed in Sect. 2.1, is part of the ASM-WB kernel. Its core is an efficient implementation of the well-known unification-based type inference algorithm [5], and can be used—even independently from the ASM interpreter—to check that ASM-SL specifications are well-typed and to infer their signatures. It also performs other simple static checks, e.g., it checks that static functions are never updated. The result of successful type checking is an abstract syntax tree (AST) with type annotations (see Fig. 1).

Simulator A simple interpreter, which allows to simulate Abstract State Machines described in ASM-SL notation, is also part of the kernel. It is based on: *(i)* an evaluator, containing functions—defined inductively over the ASM-SL abstract syntax trees—which compute values from terms and update sets from rules[10], and *(ii)* a module defining operations to manipulate runs, while keeping

[10] Essentially, the evaluator implements the denotational semantics of terms and rules.

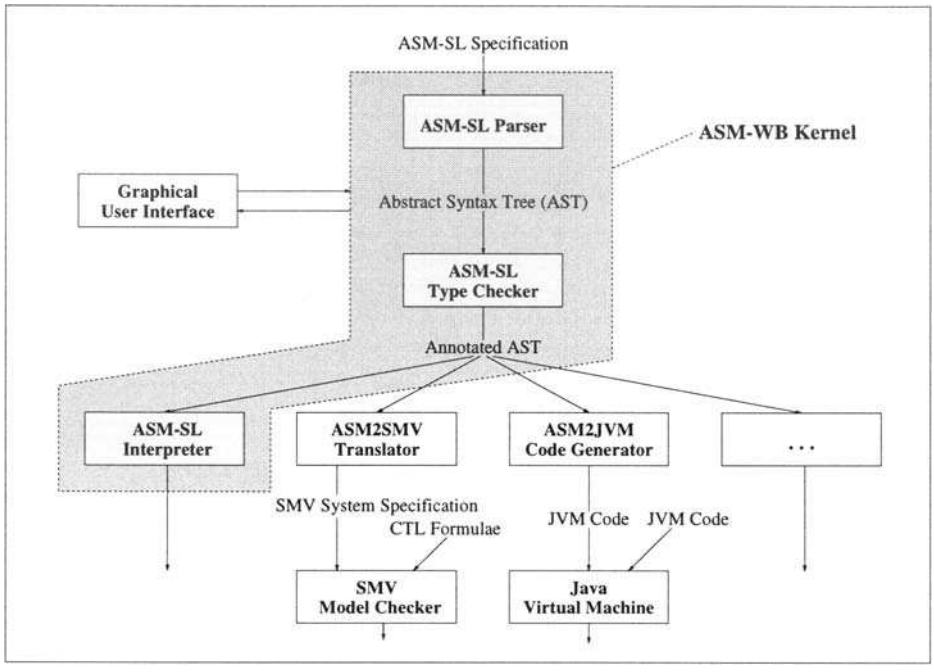


Figure1. The ASM Workbench Tool Environment

track of the computation history, such that computation steps can be retracted (*backward step* feature).

A peculiar feature of the simulator is the implementation of *external functions*: to increase the flexibility of the simulator, the mechanism to fetch values of external functions is not fixed *a priori*. Instead, this task is delegated to an external process, the so-called *oracle process*. The interaction between the oracle and the interpreter works as follows: the oracle waits for input from the interpreter; whenever the interpreter needs some external function value, it sends the function name and the arguments (in a form complying to the appropriate exchange format) to the oracle, thus activating it, and waits for the result; at this point, the oracle may perform any actions needed in order to compute or retrieve the requested value and, when finished, sends the result back to the interpreter, thus reactivating it, and waits for the next query.

Graphical User Interface Finally, the ASM Workbench provides a Graphical User Interface (GUI), which presents the information contained in ASM-SL specifications (e.g., signatures) in an orderly form and allows the user to control the simulation and inspect its results (e.g., by performing single steps forward or backward, setting breakpoints, observing the values of some terms, etc.). In this sense, the ASM Workbench's GUI can be seen as a "browser" and "debugger" of ASM specifications.

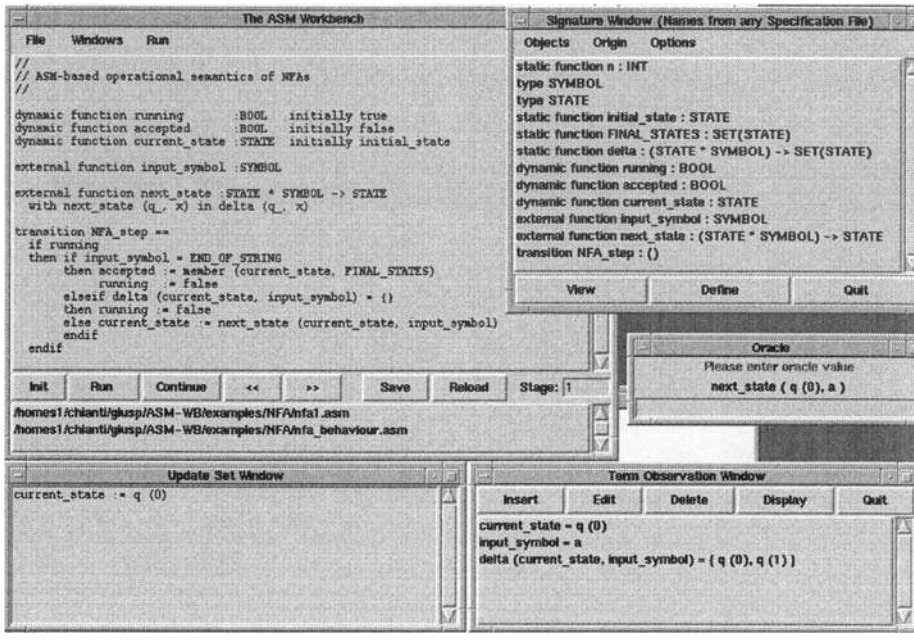


Figure2. The ASM Workbench's Graphical User Interface

The GUI also provides a default oracle, which simply invites the user to enter a value of the appropriate type. Despite its simplicity, the default oracle proves to be useful to make the first experiments with ASM models. Later, it can be replaced by more sophisticated (application-specific) oracles implementing realistic simulation scenarios.

A snapshot of the ASM Workbench's GUI is shown in Fig. 2. It is possible to see: in the main window—the large one on the left—the ASM-SL source for the ASM of Table 1 (`nfa_behaviour.asm`), the debugger controls, and the list with the specification files; in the signature window—above on the right—the whole signature, including the static specification of the NFA of Table 2 (`nfa1.asm`); in the small window below the signature, the default oracle, waiting for the user to choose a next state; in the update set window—below on the left—the update set fired in the last step; in the term observation window—below on the right—a part of the ASM state, identified by a set of terms chosen by the user.

Ongoing Work Other components which are currently under development are:

- an interface to the model-checker SMV [15], called ASM2SMV, implementing a translation of a particular class of (finite-state) ASMs into equivalent finite state machines specified in the SMV language, for which SMV can automatically check temporal properties specified by CTL formulae (see [21]);
- a code generator for the Java Virtual Machine [16], called ASM2JVM, being developed by B. Thorstensen at Tromsø University (Norway) on the top of the front-end provided by the ASM Workbench (see [7] for details).

3.3 Using, Extending, and Modifying the ASM Workbench

The ASM Workbench's resources could be deployed in different ways, namely:

1. to develop, test, validate, or analyse ASM specifications written in ASM-SL notation, by using the existing ASM-WB tools as they come: to do this, only some basic knowledge of ASM and ASM-SL and some familiarity with the tools are needed;
2. as a framework to develop other ASM-SL based tools, by reusing part of the ASM-WB functionalities: to do this, a knowledge of the interfaces (signatures) of the ASM-WB modules and the semantics of their data types and functions are needed[11]. Note that the tool integration can be done either by tight coupling or by loose coupling. Examples of integration by tight coupling are the ASM-WB's GUI and ASM2SMV, which are ML programs calling directly the kernel's functions, while the integration of ASM2JVM—a Java program whose input is constituted by the annotated ASTs exported by the type-checker—is based on loose coupling.
3. as a starting point for development of ASM tools, which do not necessarily take ASM-SL—as presented in this paper—as their source language: this could be achieved by partially reusing and partially modifying the ASM-WB modules. As an example, one could think about an untyped version of the ASM-WB or about a different type system: as the ASM-WB modules are highly orthogonal (e.g., the interpreter does not make any use of type information), it would suffice to replace the type-checker module and make a few other minor changes (e.g., change the syntax of type expressions).[12]

4 Related Work

As mentioned in the introduction, several ASM simulators (interpreters or compilers) have been developed during the last years, while there are essentially no tools supporting static analysis, transformations, verification, etc.

An early ASM simulator was developed by Angelica Kappel at Dortmund University [14]. It is based on a specification language called DASL (Dynamic Algebra Specification Language)[13], a typed specification language extending ASMs by a restricted form of multi-sorted equational specifications to define the static parts (conceptually the same approach we adopted in ASM-SL). The interpreter

[11] The corresponding documentation will be published when the kernel's interface will be stable enough, after the first experiments of ASM-WB-based tool development (ASM2SMV and ASM2JVM), which should also demonstrate the practicability of this approach, will be completed.

[12] Of course, this kind of deployment is the most advanced and difficult of the three and may possibly require too much effort for anybody who is not familiar with the internal structure and implementation of the ASM Workbench: however, we mention it for the sake of completeness.

[13] At that time ASMs were known as *Dynamic Algebras*. Then the name was changed to *Evolving Algebras* (*EAs*), and finally to Abstract State Machines.

is based on a simple abstract machine: DASL specifications are compiled into an intermediate code, which is then interpreted by the abstract machine. Both the compiler and the abstract machine are implemented in Prolog, and the user interface is basically the Prolog environment. Despite its nice features, such as the simple and clean specification language, this tool implements only a small subset of sequential deterministic ASMs, with syntax and semantics partially different from the present ASM definition (in fact, this tool was realized before [10] appeared).

Another early ASM simulator is the Michigan interpreter [12], developed at the University of Michigan at Ann Arbor. This interpreter, implemented in C, has been updated over the years and (in its most recent version) includes the complete language of ASMs as defined in [10]. The specification language is untyped and includes some constructs for the definition of the static part of ASM specifications, basically by combining a quite rich set of built-in data structures and operations: however, this part of the specification language is somewhat unsystematic and not very expressive.[14] It is also possible to "customize" the interpreter, implementing additional ASM functions as C functions: however, this is not a practical alternative for the specification of static algebras, not only because of the low abstraction level of the C language, but also because each time a new version of the interpreter has to be built. The (textual) user interface is based on the Tcl shell and allows some degree of user interaction (inspection of the interpreter state and some of the usual features found in debuggers with text-based user interface).

Related to the Michigan interpreter is another tool, a partial evaluator for ASMs implemented by Jim Huggins [11]. Unfortunately, the development of the partial evaluator was abandoned quite early and this tool did not come to maturity (to the author's knowledge, this has been the only development of an ASM tool which is not a simulator).

Two further ASM simulators are *leanEA* ("A Lean Evolving Algebra Compiler") by Posegga and Beckert [3] and the EA interpreter by Dag Diesen [6]. They are similar in the sense that, in both of them, the implementation of ASM is embedded in the implementation language (Prolog for *leanEA*, Scheme in Diesen's interpreter), on which both tools rely for the definition of the static part of specifications. Both tools support only a subset of sequential deterministic ASMs. The user interface is provided by the Prolog and Scheme environments, respectively.

A runtime system for the execution of ASMs—called ASM Virtual Architecture (ASM/VA)—was developed by Igor Đurđanović [8]. The ASM/VA is a C++ class library providing the basic mechanisms needed to make ASMs executable, to be used in combination with a code generator translating ASM specifications into C++ code containing appropriate ASM/VA calls. It is based on the C++

[14] For instance, there are lists as a built-in data structure, but no possibility of defining constructors in general; there are constructs to define static functions, but recursive definitions are not possible.

Standard Template Library (STL) and its primary aim is the efficient execution of ASMs.

The most recent ASM simulator is the *Aslan* compiler developed by Matthias Anlauff [1]. The Aslan language—which is untyped—includes all the ASM transition rules defined in [10] and some additional modularization constructs. Nondeterminism is available in the form of *choose* rules, while external functions are not supported. For defining the static parts of a specification, Aslan offers the possibility of defining constructors[15], while the set of built-in data structures and functions is very restricted: universes and static functions have to be implemented in C. Aslan generates C code, which is then linked to the Aslan run-time system and to the user-defined C functions. A special feature of Aslan, very useful when using ASMs to define the semantics of programming languages, is the possibility of integrating grammar productions in the ASM specification. The possibilities of user interaction are much more advanced than in the older tools: similarly as in the ASM Workbench, a graphical debugger allows to control the flow of the simulation, inspect the ASM state, and so on.

Conclusions

In this paper we introduced the basic ideas underlying the ASM Workbench, its language and its architecture. Our experience seems to confirm that both the language and the tools are appropriate for modelling quite different systems at quite different levels of abstraction.

Moreover, a first successful application of the ASM Workbench in an industrial context has been developed at the Munich Research Center of Siemens AG. Within the project "FALKO" [9] (dealing with a simulation system for the development and validation of train schedules) a prototype of the system was developed in ASM-SL and tested using the ASM Workbench. The size of the ASM-SL specification is quite considerable: the static part includes 323 function definitions, the dynamic part 113 transition rules. The ASM-SL specifications—in conjunction with existing libraries implementing numerical computations—have been simulated (the link to the existing code has been realized by using oracles). A test system includes 16 stations, 150 signals, 60 switches, and 5 crossings. Although the first simulations performed using the ASM Workbench's interpreter were quite slow[16] (mainly due to the large amount of interprocess communication generated by the oracle queries), a code generator, based on the ASM Workbench parser and translating the appropriate subset of ASM-SL into C++, has been later developed at Siemens, such that the same ASM-SL specifications could be simulated very efficiently.[17]

This seems to provide some evidence that the ASM Workbench constitutes a good basis for the development of ASM tools satisfying the requirements of real-life specification and modelling tasks.

[15] The untyped counterpart of freely generated types.

[16] About 24 hours simulation time for 4,5 hours train schedule.

[17] About 2-5 minutes for the same train schedule.

The current version of the ASM Workbench and of its documentation can be found at

`http://www.uni-paderborn.de/cs/asm/ASMToolPage/asm-workbench.html`.

References

1. M. Anlauff. Aslan: Programming in Abstract State Machines (draft of the language specification). Available at `http://www.first.gmd.de/~ma/projects.htm`, 1998.
2. M. Anlauff, P. Kutter, and A. Pierantonio. Formal Aspects of and Development Environments for Montages. In M. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Workshops in Computing, Amsterdam, 1997. Springer.
3. B. Beckert and J. Posegga. leanEA: A Lean Evolving Algebra Compiler. In H. Kleine Büning, editor, Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL'95), volume 1092 of *LNCS*, pages 64–85. Springer, 1996.
4. G. Del Castillo. ASM-SL, a Specification Language based on Gurevich's Abstract State Machines: Introduction and Tutorial. Technical report (to appear), Universität-GH Paderborn, 1999.
5. L. Damas and R. Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.
6. D. Diesen. *Specifying Algorithms Using Evolving Algebra. Implementation of Functional Programming Languages.* Dr. scient. degree thesis, Dept. of Informatics, University of Oslo, Norway, March 1995.
7. D. Diesen, T.O. Svendsen, and B. Thorstensen. Developing new ASM tools. In U. Glässer and P.H. Schmitt, editors, *Proceedings of the Fifth International Workshop on Abstract State Machines*, pages 155–158, Magdeburg, Germany, September 1998.
8. I. Durdanovic. *From Operational Specifications to Real Architectures (work in progress).* PhD thesis, Universität-GH Paderborn, to appear in 1999.
9. FALKO—Fahrplanvalidierung und Konstruktion in Nahverkehrssystemen. Siemens München, ZT SE 4.
10. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
11. Y. Gurevich and J. Huggins. Evolving Algebras and Partial Evaluation. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 587–592, Elsevier, Amsterdam, the Netherlands, 1994.
12. J. Huggins and R. Mani. The Evolving Algebra Interpreter Version 2.0. Documentation of the Michigan Evolving Algebra Interpreter, available electronically at `ftp://ftp.eecs.umich.edu/groups/Ealgebras/interp2.tar.Z`.
13. C.B. Jones. *Systematic Software Development using VDM.* Prentice Hall, 1990.
14. A.M. Kappel. Executable Specifications Based on Dynamic Algebras. In A. Voronkov, editor, *Logic Programming and Automated Reasoning*, volume 698 of *Lecture Notes in Artificial Intelligence*, pages 229–240. Springer, 1993.
15. K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.
16. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

17. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.
18. S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer-Verlag, 1992.
19. W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications.* Kluwer Academic Publishers, 1998.
20. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science*, 3(4):377–413, 1997.
21. K. Winter. Model Checking for Abstract State Machines. *Journal of Universal Computer Science*, 3(5):689–701, 1997.
22. W. Zimmerman and T. Gaul. On the Construction of Correct Compiler Back-Ends: An ASM Approach. *Journal of Universal Computer Science*, 3(5):504–567, 1997.

The IFAD VDM Tools:

Lightweight Formal Methods

Sten Agerholm and Peter Gorm Larsen

IFAD, Forskerparken 10
DK-5230 Odense M, Denmark
E-mail: toolbox@ifad.dk Web: http://www.ifad.dk
Telephone: +45 63 15 71 31 Fax: +45 65 93 29 99

Abstract. The services and tools supporting the ISO Standard VDM-SL notation and its object-oriented extension VDM++ are commonly known as the VDM Technology. For both notations the company IFAD provides leading edge technology tools, training and consultancy. Users of the VDM Technology typically report on their work in the Toolbox Newsletter which is issued on a regular basis. This note provides a brief overview of the capabilities of the tools.

1 IFAD Profile

IFAD is a world-leading software development technology provider, situated in the International Science Park Odense, Denmark:

Tool Vendor: IFAD provides professional software development tools that assist engineers in producing high-quality software.
Service Provider: IFAD ensures technology transfer by offering training courses customer specific consultancy, and by organizing industrial seminars.
Subcontractor: IFAD offers subcontracted requirements specification and software development performed by highly qualified and experienced personnel.

IFAD has extensive experience in technology transfer projects with a range of major European companies. Our list of references includes, for example, Adelard (UK), Aerospatiale (F), Alenia (I), Ansaldo (I), Baan Front Office Systems (DK), British Aerospace Systems and Equipment (UK), Dassault Electronique (F), DDC-I (DK), Formal System Inc. (CAN), GAO (D), Japan Railways (JPN), and Origin (NL).

2 VDM Tools: *Validated Design through Modeling*

An overview of the IFAD VDM Tools is presented in Fig. 1, which illustrates how a number of facilities are integrated via a specification manager. Powerful analysis and consistency checking tools help the designer to achieve the highest confidence in system designs. There is a bi-directional coupling module for the wide-spread graphical notation UML in order to ensure the optimal support for complementary graphical and textual notations such as UML and VDM++.

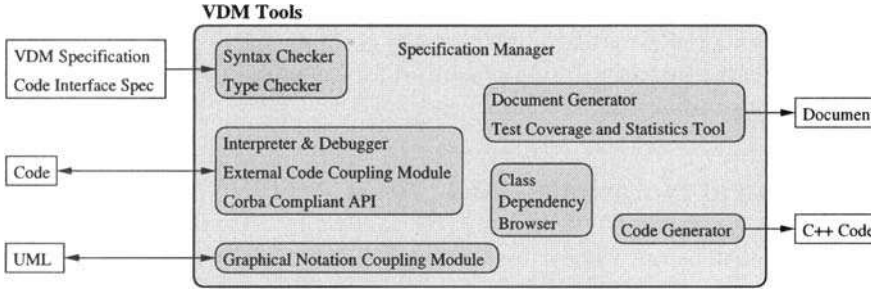


Fig. 1. Overview of the IFAD VDM Tools.

3 Executable Models: *Clarify Requirements and Find Defects*

A key facility of VDM Tools with many benefits is an interpreter and interactive debugger for executing and debugging high-level system models written in the VDM notations. For example, this supports that system requirements and hidden assumptions can be clarified by applying rapid prototyping techniques in order to validate system models. This results in early trouble shooting and thereby reduces development costs and time-to-market. Furthermore, requirement properties can be documented explicitly by annotating VDM specifications with predicates that are verified automatically while executing test cases. Finally, inspections can be automated by this type of specification testing, thereby making these cheap to conduct and not subject to human errors nor limitations. Moreover, inspections can be repeated at no cost, which is particularly advantageous in situations with changing requirements.

Executable models are also essential from a technology transfer perspective. Software engineering practice is heavily based on testing and so this technique is well-known to engineers. VDM Tools support the testing process by providing test coverage and test statistics information at model level and through facilities for executing external code together with models: the Corba compliant API facility and the external code dynamic link facility. These support rapid prototyping, for example, by building graphical front-ends for systems designed in VDM. Hence, it is possible to get early feedback on a model from a customer, manager or colleague, who may not be familiar with VDM.

4 Code Generation: *Reduce Cost and Time-to-market*

Automatic code generation helps to solve the consistency problem between specification and implementation and eliminates the human factor concerning the potential introduction of errors during implementation. Hence, automatic code generation can help to minimize development costs and time-to-market. Moreover, bug fixing and other maintenance can be performed at the "high" specification rather than the "low" implementation level, yielding easier maintenance and lower costs.

The IFAD VDM tools support automatic generation of C++ code from high-level models as an add-on feature. The tools generate fully executable code for 95% of all VDM constructs leaving facilities for including user defined code for non-executable parts of the specification.

5 Graphical Notations: *Master Complexity*

Graphical notations are widely used for object-oriented designs and embedded real-time systems. The Rose-VDM++ Link offers the complementary benefits of the textual notation VDM++ and the graphical notation UML for object-orientation through a bi-directional link to Rational Rose. Hence, the link supports round trip engineering where an engineer can begin the development process by writing graphical models in UML, automatically translate these to VDM++, incrementally develop more concise models, test the models using the VDM++ interpreter, change class structure, merge with existing UML, make changes in UML, merge with existing VDM++, etc. The advantage of using VDM++ is that it allows more concise system specifications than with purely graphical notations and that these can be executed and debugged early in the development process.

6 Facilities

The facilities presented in Fig. 1 are described in more detail below.

Specification Manager The Specification Manager maintains a project by keeping track of the status of modules (VDM-SL) or classes (VDM++). These status indications present an easy overview of the model configuration, as a system model may be distributed over several files. Furthermore, the Specification Manager can automatically update the specification with respect to syntax checking, type checking, code generation, etc. Finally, the user has a number of options, for example, he can select his favorite editor.

Interpreter The VDM interpreters support all executable constructs in VDM-SL and VDM++. These range from simple value constructors like set comprehension and sequence enumeration to more advanced constructs like exception handling, lambda expressions, loose expressions and pattern matching. One of the benefits of executing specifications is that testing techniques can be used to assist validation of specifications. In the development process small or large parts of a specification can be executed to enhance the designer's knowledge of and confidence in the specification. Furthermore, an executable specification can form a running prototype.

Debugger A source-level debugger is essential when working with executable specifications. The VDM-SL debugger supports the functionality found in debuggers for ordinary programming languages including: setting breakpoints, stepping, inspection of all variables defined in the scope, and inspection of the call stack.

Type Checker The static semantic analyzer is an advanced type checker and it supports most of the static semantics levels prescribed by the ISO Standard. It contains a powerful type inference mechanism which also shows proof obligations with respect to the type system.

Test Facility Test coverage information can be automatically recorded during the evaluation of a test-suite. The specifier can at any point view which parts of the specification are most frequently evaluated and which parts have not been covered at all. The test coverage information is displayed directly in the source file in a comprehensive form which is easy to read. The source file can be a Microsoft Word or LaTeX document.

Automatic Code Generator The IFAD VDM Tools support automatic generation of C++ code from a VDM-SL or VDM++ specification which helps to solve the consistency problem between specification and implementation. The code generator produces fully executable code for 95% of all VDM-SL constructs leaving facilities for including user defined code for non-executable parts of the specification.

Dynamic Link Facility The IFAD VDM-SL Toolbox has an add-on feature which makes it possible to integrate external code into the execution of a specification. This can be used to integrate a formal model with components developed in a traditional way and provide graphical front-ends for a model.

Corba Compliant API The IFAD VDM Tools provide a Corba compliant API which allows other programs to access a running Toolbox. This enables external control of the Toolbox components such as the type checker, interpreter and debugger. The API allows any code such as a graphical front-end or existing legacy code to control a Toolbox.

Rose-VDM++ Link The Rose-VDM++ Link integrates UML and VDM++. Through bi-directional translations the link provides a tight coupling of the IFAD VDM++ Toolbox and Rational Rose. Hence the link supports round trip engineering between UML and VDM++, where the graphical notation is used to provide the structural, diagrammatic overview of a model while the formal notation is used to provide the detailed functional behavior of a model.

7 Platforms

The IFAD VDM Tools run on PCs under Windows and Linux and on a number of UNIX platforms.

KIV 3.0 for Provably Correct Systems

Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel

Abt. Programmiermethodik
Universität Ulm, D-89069 Ulm, Germany
email: {balser,reif,schellhorn,stenzel}@informatik.uni-ulm.de

1 Synopsis

KIV 3.0 is an advanced tool for engineering high assurance systems. It provides an economically applicable verification technology, and supports the entire design process from formal specifications to executable verified code. In KIV the design process for high assurance systems proceeds as follows.

1. KIV supports both functional and state based software/system design using algebraic specifications or Abstract State Machines (ASMs), respectively. As a first step, predefined theories from a library can be imported. New specifications are added to the hierarchically structured specification graph which is graphically visualized.
2. In addition to the specification, a formal safety/security model is defined. The formulation of extra validation properties helps to detect gross specification errors before it is attempted to prove the main safety/security properties.
3. It has to be shown that the validation and safety/security properties are satisfied by the specification. The necessary formal proofs are done in an interactive graphical proof environment. Proof search is automated to a large extent. Proof engineering facilities help to reveal specification errors. After correcting the specification, invalid proofs can be reused automatically.
4. The components of the hierarchical system specification can be implemented independently (modular) using an imperative programming language. Proof obligations for the correctness of the implementation are generated automatically and have to be verified by the proof component. Again, corrected errors lead to invalidated proofs which can be reused automatically.
5. The whole specification and verification process is guarded by an elaborate correctness management. If, finally, every specification and implementation is in "proved state", it guarantees that there are no inconsistencies and all proof obligations and used lemmas are proved.
6. For use in future projects, specifications and implementations can be added to a library.

KIV 3.0 is ready for use and has been tested in a number of industrial pilot applications. However, it can also be used as a pure specification environment with a proof component. Furthermore, KIV serves as an educational and experimental platform in formal methods courses. Details on KIV can be found in [17] [18] [19] and under http://www.informatik.uni-ulm.de/pm/kiv/kiv.html.

Related interactive systems with major applications in the area of correct software/system designs are ACL2 [13], PVS [14], ISABELLE [15], HOL [7], or VSE [11].

2 Structured Specifications and Implementations

For functional modelling, KIV relies on first-order algebraic specifications to describe hierarchically structured systems in the style of ASL [21]. Specifications are built up from elementary first-order specifications with the operations enrichment, union, renaming, parameterization and actualization. Specifications have a loose semantics and may include generation principles to define inductive theories. State-based or reactive systems can be modelled with Abstract State Machines [9] [22], where used data types are again specified algebraically. These formalisms can be used to formulate requirements, to describe (arbitrary) systems or processes, or – more specifically – to specify software systems.

When used for a software system, KIV supports the stepwise refinement and implementation of the specification. Specification components can be implemented using program modules in an imperative programming language. The designer is subject to a strict decompositional design discipline leading to modular systems with compositional correctness. As a consequence, the verification effort for a modular system becomes linear in the number of its modules [16]. The correctness of each single module guarantees that the whole implementation fulfills its specification.

Structured specifications and implementations form a development graph. For each component, a theorem base exists where theorems are stored. The proof of the theorem is done relative to its component, and may use theorems of other components lower in the specification hierarchy. A lemma needed in a proof is added to the theorem base of the lowest component in the specification hierarchy where it is valid. This allows the reuse of the lemma in other proofs in other specifications as well. The hierarchical approach makes it feasible to deal with large specifications with thousands of theorems.

3 Interactive Theorem Proving

KIV offers an advanced interactive deduction component based on proof tactics. It combines a high degree of automation with an elaborate interactive proof engineering environment. KIV can be used for specification validation and design verification as well as for program verification. The interactive proof strategy is based on a sequent calculus. For first-order reasoning, the proof tactics include rewriting, forward reasoning and induction. For the verification of implementations with imperative programs, the proof strategy is based on symbolic execution and induction using Dynamic Logic [6] [10].

To automate proofs, KIV offers a number of heuristics, see [18]. Among others, heuristics for induction, unfolding of procedure calls, and quantifier instantiation are provided. Heuristics can be chosen freely, and the choice is alterable any

time during the proof. So called "problem specific" heuristics are easily adaptable to specific applications without changing their implementation. Usually, the heuristics manage to find 80 – 100 % of the required proof steps automatically. First order goals can be forwarded to external automatic theorem provers [2].

KIV's simplifier handles hundreds and even thousands of conditional rewrite rules very efficiently, using discrimination nets and a compilation technique [8] [12] with some extensions like AC-rewriting and forward reasoning. As the structure of a formula assists in understanding its meaning, it is essential that simplification in KIV is structure preserving. Rewrite and simplification rules are explicitly chosen by the user. For standard data types, which are stored in a library, sets of predefined simplifier rules are available for reuse in further projects.

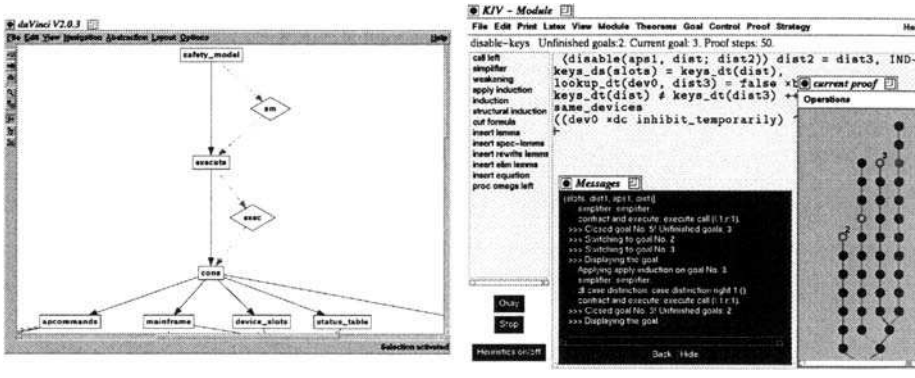


Fig. 1. Two snapshots of the system

4 Proof Engineering

Frequently, the problem in engineering high assurance systems is not to verify proof obligations affirmatively, but rather to interpret failed proof attempts indicating errors in specifications, programs, lemmas etc. Therefore, KIV offers a number of proof engineering facilities to support the iterative process of (failed) proof attempts, error detection, error correction, and re-proof. Dead ends in proof trees can be cut off, proof decisions may be withdrawn both chronologically and non-chronologically. Unprovable subgoals can be detected by automatically generating counter examples. Another interesting feature of KIV is its strategy for proof reuse. Both successful and failed proof attempts are reused automatically to guide the verification after correction [20]. This goes beyond proof replay (or proof scripts). We found that typically 90% of a failed proof attempt can be reused for the new proof after a correction.

KIV offers a powerful graphical interface (see Fig. 1): The dialog is menu oriented, the structure of specifications and implementations is visualized as a

hierarchical graph using daVinci [4]. Proofs are presented as trees (with colored nodes), where the user can click on nodes to inspect single proof steps. Proof trees are permanently stored in a very efficient manner, so that a partial proof can be continued at a later time. The system can generate LaTeX output of specifications, theorems, and proofs, which is very helpful for documentation purposes. Furthermore, statistical data (about lines of specification, number of axioms, proofs, proof steps etc.) is computed automatically.

5 Correctness Management

One of the distinguished features of KIV is its correctness management. It ensures that no cyclical proof dependencies exist, and that after modifications to specifications, implementations or theorems only those proofs are invalidated that are really affected by the modification. We distinguish between local and global correctness management.

Local correctness. A theorem base stores all axioms, theorems, and proof obligations for one specification component or implementation module. Local correctness management is concerned with one theorem base only. KIV does not enforce a bottom-up proof strategy. This means that theorems can be used in a proof before they are proved themselves. Of course, it is illegal to use a theorem A as a lemma in the proof of B and at the same time B in the proof of A. To achieve this goal, it is necessary to keep track of each used lemma, rewrite rule, and simplification rule in a proof. A using graph contains the information *which theorem uses which other theorems. This using graph must be acyclic.* When beginning a proof, all theorems that (transitively) use this theorem are "hidden", i.e. they cannot be used as lemmas or rewrite rules. This approach is much more efficient than cycle checking before each application of a theorem, and, of course, much better than checking after a proof is finished. Another task of the local correctness management is to deal with the deletion or modification of a theorem and the effects on other proofs in the same theorem base. A theorem can be modified even if it is used as a lemma in a proof of another theorem. The proofs for both theorems will become invalid, which means that both theorems are considered unproved. However, the invalid proofs are kept for reuse. If all theorems of a theorem base are proved and their proofs are valid, the theorem base is locally correct.

Global correctness. A theorem L (e.g. a rewrite rule) may be used in many proofs in different theorem bases. If the theorem is modified or deleted, these proofs must be invalidated. However, KIV does not automatically store the information in which proofs a theorem is used, but for each proof which theorems it uses. The reasons are efficiency considerations and the additional problems caused by redundant information. Again for efficiency considerations, it is a bad idea to inspect every proof of every theorem base that is based on L. This could involve the inspection of hundreds of proofs.

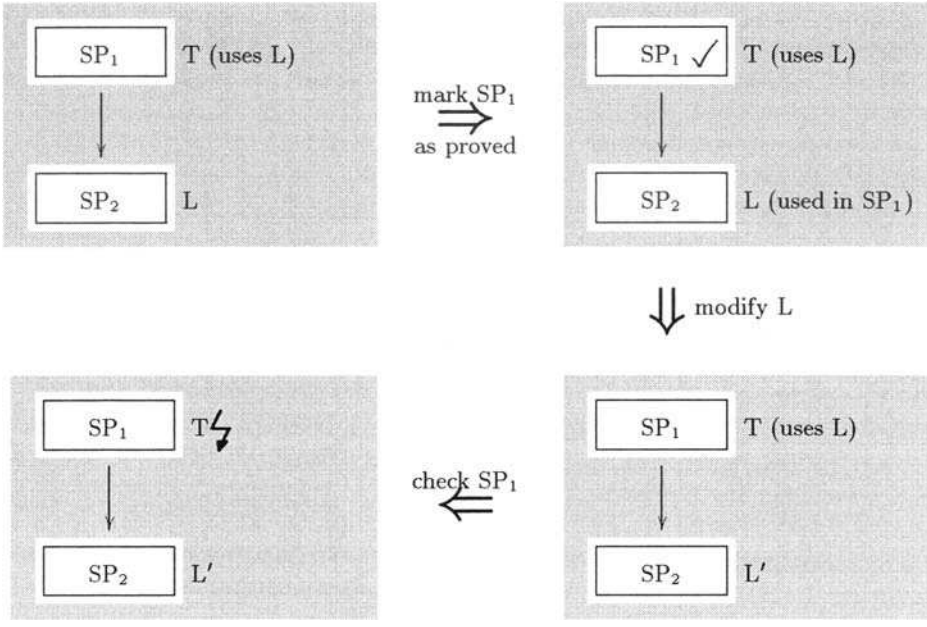


Fig. 2. Correctness Management

KIV delays the invalidation of these proofs and uses the concept of "proved states" for theorem bases. A theorem base can enter proved state if it is locally correct (i.e. all theorems are proved and all proofs are valid) and all theorems from other bases used in some proof exist (and are unchanged). Then some redundant information is recorded. In each theorem base the list of theorems that are used in some proof of the current base (that enters proved state) is stored. If a used theorem is modified or deleted this redundant informations is used to remove the base from proved state. A theorem base in proved state may not be modified. For example, adding a new theorem automatically removes it from proved state.

Consider a situation where a theorem T in SP_1 uses a lemma L from SP_2 (as depicted in the upper left corner in Fig. 2). Entering proved state for SP_1 is initiated by the user. Then the above mentioned checks are performed, and in SP_2 the information is stored that L is used in SP_1 (upper right corner in Fig. 2). It is not stored by which theorems of SP_1 L is used. Finally, SP_1 is marked as proved. If L is modified, SP_1 is removed from the proved state using the information stored in SP_2. This leads to a situation where the proof for T is not marked as invalid but in fact is invalid (lower right corner in Fig. 2). Of course, trying to enter proved state again will fail. The user can run a check on SP_1 that marks T as invalid (lower left corner in Fig. 2). This approach allows complete control over the time when complex and expensive operations are performed. The main point is that for a theorem base in proved state everything is guaranteed correct. If all theorem bases are in proved state, everything is globally correct.

6 Some Applications

Access Control. In this case study, a generic access control model for a computer system (based on [1]) was specified, implemented, and the implementation was proved correct. Furthermore, it was formalized and proved that it is not possible for a user to increase his rights without help from others. All specifications together contain about 1100 lines of text, while the efficient implementation has a size of 1200 lines of text. All in all 837 theorems and lemmas were proved. The overall time needed to complete the case study (including a vast number of modifications, error corrections, and reuses of proofs) was 14 weeks. See [5].

Digital Signatures.[1] In 1997, German Parliament passed the "Signaturgesetz" which is a legal basis for the usage of digital signatures. In order to receive a certificate for critical parts of the software and organization structures related to the application of digital signatures, a formal security model has to be provided. In this case study, it has been shown that it is possible to develop such a formal security model. The process of creating and verifying digital signatures has been formally specified. The integrity and nonrepudiation of signatures has been proved. Overall 800 lines of specification were needed.

Airbag. The decision to fire an airbag in a car is done by software in a black box. While the algorithm is very short (about two pages of text), the software runs on a micro controller with only 16bit integers. Together with the company manufacturing the boxes we formally verified that the algorithm fulfills its specification and never produces an arithmetical overflow. Another result are restrictions on the used parameters that must hold to avoid an arithmetical error. If the parameters are changed (for another type of car), only these restrictions have to be checked again.

Compiler Verification. In [3], the compilation of PROLOG into code for the Warren Abstract Machine (WAM) is described with 11 transformation steps using Abstract State Machines (ASMs, [9]). Meanwhile, we have formalized and verified 6 transformation steps (with 1500 lines of specification and 700 lines of code). The most complex verification step requires an invariant which covers about three pages of text. During the verification, several errors were revealed in the compiler assumptions as well as in the interpreters. See [22].

Safe Command Transfer in a GNC.[1] In cooperation with the company that developed the software (intecs sistemi, Pisa), part of the guidance and navigation control (GNC) system of a spacecraft was developed formally and reevaluated in KIV 3.0. The given safety requirements have been verified, and a prototypical implementation has been proved correct. The major benefits of the formal

[1] Part of the BSI (Bundesamt für Sicherheit in der Informationstechnik) project VSE (Verification Support Environment). The prover of the VSE tool is based on KIV 1.0 and INKA.

verification were the detection of a missing case in the informal specification (based on ESA Software Engineering Standards), and the explicit (and correct) specification of implicit assumptions about the input parameters.

Dynamic Hash Tables. This case study is concerned with the verification of a store structure by dynamic (or extendable) hash tables that never need to be reorganized, even when an arbitrary number of elements is inserted, and although collision lists of bounded size are used. The verification is extraordinary difficult due to the complexity of the insert- and delete-procedure (which use many auxiliary procedures) and to the heavy use of arithmetic (including exponentiation and modulo). Details can be found in [5].

A Library of Reusable Specifications. The reuse of standard data types decreases the time needed to develop the first version of a new structured specification considerably. The specifications are correct, and contain a large set of already proved properties and rewrite rules which increases over time. Our library currently contains specifications for 37 data types with 297 functions and 1977 proved lemmas.

References

1. M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A Calculus for Access Control in Distributed Systems. In J. Feigenbaum, editor, *CRYPTO '91*. Springer LNAI 576, 1991.
2. W. Ahrendt, B. Beckert, R. Hähnle, W. Menzel, W.Reif, G. Schellhorn, and P. Schmitt. Integrating of Automated and Interactive Theorem Proving. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume I: Foundations, chapter 3: Interactive Theorem Proving. Kluwer Academic Publishers, 1998.
3. E. Börger and D. Rosenzweig. The WAM—definition and compiler correctness. In Christoph Beierle and Lutz Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, volume 11 of *Studies in Computer Science and Artificial Intelligence*. North-Holland, Amsterdam, 1995.
4. M. Fröhlich and M. Werner. Demonstration of the interactive graph visualization system *daVinci*. In R. Tamassia and I. Tollis, editors, *DIMACS Workshop on Graph Drawing '94. Proceedings*, Springer LNCS 894. Princeton (USA), 1994. http://www.informatik.uni-bremen.de/~davinci/.
5. T. Fuchß, W. Reif, G. Schellhorn, and K. Stenzel. Three Selected Case Studies in Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*. Springer LNCS 1009, 1995.
6. R. Goldblatt. *Axiomatising the Logic of Computer Programming*. Springer LNCS 130, 1982.
7. M. Gordon. HOL: A Proof Generating System for Higher-order Logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification and Synthesis*. Kluwer Academic Publishers, 1988.
8. P. Graf. *Term Indexing*. Springer LNCS 1053, 1996.

9. M. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods.* Oxford University Press, 1995.
10. M. Heisel, W. Reif, and W. Stephan. A Dynamic Logic for Program Verification. In A. Meyer and M. Taitslin, editors, *Logical Foundations of Computer Science.* Springer LNCS 363, 1989.
11. D. Hutter, B. Langenstein, F. Koob, W. Reif, C. Sengler, W. Stephan, M. Ullmann, M. Wittmann, and A. Wolpers. The VSE Development Method – A Way to Engineer High-Assurance Software Systems. In Bredereke Gotzheim, editor, *GI/ITG Tagung Formale Beschreibungstechniken für verteilte Systeme.* Universität Kaiserslautern, 1995.
12. S. Kaplan. A compiler for conditional term rewriting systems. In *2nd Conf. on Rewriting Techniques anf Applications. Proceedings.* Bordeaux, France, Springer LNCS 256, 1987.
13. M. Kaufmann and J Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, 23(4), April 1997.
14. S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *Automated Deduction - CADE-11. Proceedings*, Springer LNAI 607. Saratoga Springs, NY, USA, 1992.
15. L. C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer LNCS 828, 1994.
16. W. Reif. Verification of Large Software Systems. In R. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science. Proceedings.* Springer LNCS 652, 1992.
17. W. Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report.* Springer LNCS 1009, 1995.
18. W. Reif, G. Schellhorn, and K. Stenzel. Interactive Correctness Proofs for Software Modules Using KIV. In *Tenth Annual Conference on Computer Assurance*, IEEE press. NIST, Gaithersburg (MD), USA, 1995.
19. W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications.* Kluwer Academic Publishers, 1998.
20. W. Reif and K. Stenzel. Reuse of Proofs in Software Verification. In R. Shyamasundar, editor, *Foundation of Software Technology and Theoretical Computer Science. Proceedings.* Springer LNCS 761, 1993.
21. D. T. Sannella and M. Wirsing. A kernel language for algebraic specification and implementation. In *Coll. on Foundations of Computation Theory*, Springer LNCS 158. Linköping, Sweden, 1983.
22. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science (J.UCS)*, 1997. available at http://hyperg.iicm.tu-graz.ac.at/jucs/.

PVS: An Experience Report*

S. Owre, J. M. Rushby, N. Shankar, and D. W. J. Stringer-Calvert

Computer Science Laboratory, SRI International, Menlo Park CA 94025 USA
{owre, rushby, shankar, dave_sc}@csl.sri.com
URL: http://pvs.csl.sri.com

Abstract. PVS is a comprehensive interactive tool for specification and verification combining an expressive specification language with an integrated suite of tools for theorem proving and model checking. PVS has many academic and industrial users and has been applied to a wide range of verification tasks. In this note, we summarize some of its applications.

1 Introduction to PVS

PVS (Prototype Verification System) is an environment for constructing clear and precise specifications and for efficient mechanized verification. It is designed to exploit the synergies between language and deduction, automation and interaction, and theorem proving and model checking. The PVS specification language is a typed higher-order logic with a richly expressive type system with predicate subtypes and dependent types. Typechecking in this language requires the services of a theorem prover to discharge proof obligations corresponding to subtyping constraints.

The development of PVS began in 1990, and it was first made publicly available in 1993. Subsequent releases have increased its robustness and speed, and added a host of new capabilities. The essential features of PVS have already been described in prior publications [31, 33, 42], and comprehensive details can be found in the system manuals that are available from the PVS web site at http://pvs.csl.sri.com. In this note, we indicate the capabilities of the system through a survey of some of the applications for which it has been used. Due to space constraints, this is only a small sampling of the applications that have been performed using PVS, and even those that are mentioned are often given without full citations (we generally cite only the most accessible and the most recent works). We apologize to all PVS users whose work is omitted or mentioned without citation, and refer all readers to the online PVS Bibliography for a comprehensive list of citations to work concerning PVS [39].

We divide PVS activities and applications into a few broad subject areas: library development, requirements analysis, hardware verification, fault-tolerant algorithms, distributed algorithms, semantic embeddings/backend support, real-time and hybrid systems, security and safety, and compiler correctness.

* The development of PVS was funded by SRI International through Internal R&D funds. Various applications and customizations have been funded by NSF Grants CCR-930044 and CCR-9509931, and by contracts F49620-95-C0044 from AFOSR, NAS1-20334 from NASA, and N00015-92-C-2177 from NRL.

2 PVS Library Development

A major cost in undertaking formal specification and verification is that of developing formalizations for all the "background knowledge" that is required. PVS libraries help reduce this cost by providing formalizations for many common mathematical domains. Good libraries are challenging to develop: not only must they provide foundational definitions and axiomatizations that are correct, together with a body of derived constructions and lemmata that are rich enough to support development of clean, succinct, and readable specifications, but they must express these in a way that allows the PVS theorem prover to make effective use of them.

The "prelude" library built in to PVS provides many useful definitions and theorems covering basic mathematical concepts such as sets, bags, functions, relations, and orderings, together with properties of real and integer arithmetic outside the domain of the PVS decision procedures (principally those involving nonlinear arithmetic).

External PVS libraries provide finite sets, floor and div/mod, bitvectors, coalgebras, real analysis, graphs, quaternions, μ-calculus, and linear and branching time temporal logics. Development of libraries is very much a community effort in which sharing, modification, and extension has allowed the PVS libraries to grow into effective, robust and reusable assets. For example, the library for undirected graphs was developed by NASA Langley to support a proof of Menger's theorem [7]. This was extended to directed graphs by the University of Utah to support analysis of PCI bus transactions [29], and subsequently re-adopted and generalized by NASA.

3 Requirements

There is extensive evidence that requirements capture is the most error-prone stage in the software engineering lifecycle, and that detection and removal of those errors at later stages is very costly. Requirements provide a fruitful application area for formal methods because relatively "lightweight" techniques have proved effective in detecting numerous and serious errors. PVS supports these activities by providing direct support for consistency and completeness checking of tabular specifications [32], and through the process of "formal challenges" [40] where expected properties are stated of a specification and examined by theorem proving or model checking.

PVS has been used by multiple NASA centers to analyze requirements for the Cassini Spacecraft [14] and for the Space Shuttle [10], and by the SafeFM project (University of London) in the analysis of requirements for avionics control systems [13].

4 Hardware Verification

Applications of PVS to hardware verification fall into two broad classes. One class is concerned with verification of the complete microarchitecture against

the instruction set architecture seen by machine code programmers. While the presence of pipelining and other optimizations introduces complexities, the basic approach to this class of verifications depends on efficient symbolic simulation and equality reasoning, which in PVS are achieved by its tight integration of cooperating decision procedures with rewriting, combined with BDD-based simplification. PVS has been used for the full or partial verification of microcoded avionics and Java processors developed by Rockwell Collins [19], as well as for a number of smaller DLX-like processors with complex pipelines.

The other class of hardware applications concerns the complex circuits, algorithms, and protocols that are the building blocks of modern processors; these applications are sufficiently difficult that success depends on finding an effective methodology. Examples include verification of SRT dividers and other arithmetic circuits at NASA [28] and SRI, out-of-order execution at the University of Utah and SRI [24] and the Weizmann Institute [37], and cache-coherence at Stanford University [34]. Some applications are best handled using a combination of tools; PVS was used in this way by Fujitsu for the validation of the high-level design of an ATM switch [38].

5 Fault-Tolerant Algorithms

Mechanisms for fault tolerance are a significant component of many safety-critical systems: they can account for half the software in a typical flight-control system, and are sufficiently complicated that they can become its primary source of failure! Verifications of practical fault-tolerant designs are quite difficult and are often achieved incrementally, as more real-world complexities are layered on to a basic algorithm. The parameterized theories and strict dependency checking of PVS help in these incremental constructions.

For example, formal analysis of Byzantine fault tolerant clock synchronization has been elaborated over nearly a decade, with contributions from SRI and NASA Langley (using a predecessor to PVS) and the University of Ulm, culminating in verification of the algorithm used in a commercial system for safety-critical automobile control [36]. Comparable developments at SRI, NASA, Allied Signal, and Ulm have verified practical algorithms for consensus, diagnosis, and group membership, together with overall architectures for state machine replication and time-triggered execution of synchronous algorithms.

6 Distributed Algorithms

The fault tolerance applications described above employ synchronous algorithms. Other distributed algorithms are often asynchronous and are generally modeled as transition relations. Safety properties are traditionally verified by invariance arguments, and generation of suitably strong invariants is the major methodological challenge. More recent approaches employ abstraction to a finite-state (or other tractable) system that can be model checked. PVS has a model checker integrated with its theorem prover, so that it is able to perform all the stages of

such approaches. Examples include communications protocols [20] and garbage collection algorithms, parallel simulation algorithms [46] and parallelizing techniques [9], and operating system buffer-cache management [35].

Current research focusses on methods for automating the generation of abstractions and invariants [1,5,43].

7 Semantic Embeddings and Backend Support

For some applications it is convenient to use a customized logic for both specification and reasoning. Such logics can be encoded in the higher-order logic of PVS using either shallow or deep semantic embeddings. Examples include the Duration Calculus [44], DisCo [27], the B method [30], and coalgebraic treatments of Java classes [26]. An advantage of these embeddings over dedicated verification support is that the full expressiveness and power of PVS is available for all the auxiliary concepts and data types that are required.

An API for semantic embeddings of other logics is currently under development; this will allow specifications and proofs to be presented directly in the notation of the embedded logic.

An alternative to semantic embedding is to use PVS to discharge proof obligations generated by the support tool for another language. This route has been explored at Michigan State [21] and Bremen [6] universities.

8 Real-Time and Hybrid Systems

Formal treatments of real-time systems often employ special temporal or Hoare logics. Some of these have been supported by semantic embedding in PVS, as described above; others include timed automata [4], the language Trio [2], and the compositional method of Hooman [23]. Applications include several standard test-pieces, such as the Fischer's mutual exclusion algorithm, the Generalized Railroad Crossing, and the Steam Boiler, as well as some realistic protocols.

A real-time kernel for supporting Ada95 applications on a uniprocessor embedded system has also been developed in PVS at the University of York [16].

9 Security and Safety

Strong protection of data belonging to different processes is required for both security and safety in several applications. A formulation of this property in terms of "noninterference" forms one of the PVS tutorial examples. More elaborate and realistic treatments based on the same idea have been developed for security at Secure Computing Corporation [22], and for safe "partitioning" in avionics at NASA [47] and Rockwell Collins [50].

Ongoing work at SRI is developing an efficient approach for the verification of cryptographic protocols, while the special security problems arising in active networks have been formalized at the University of Cincinnati [12].

10 Compiler Correctness

In most system developments, correctness of the translation from source code to object code is not a source of major concern. Testing is performed on object code, which is fortuitously effective in finding errors introduced during compilation, assembly and linking. For critical developments, however, further assurance may be required.

PVS has been used to perform the verification of a compiler for a small safety critical language [45], and to reason about object code in terms of flow graphs [49]. The Verifix project (http://i44s11.info.uni-karlsruhe.de/ verifix/) at the Universities of Karlsruhe, Kiel, and Ulm has verified several compilation and optimization algorithms (including some expressed as abstract state machines, ASMs, where errors were found) and has also developed a collection of PVS theories for reasoning about operational and denotational semantics in this context. Another application related to programming language implementation is the security of Java style dynamic linking [11].

11 Summary

The applications sketched above give an idea of the range of projects for which PVS has been used and also provide a resource for those undertaking similar work. Additional descriptions can be found in the PVS Bibliography, which provides over 250 citations [39].

The development of PVS has been strongly influenced by practical applications and by feedback from users, and we expect this to continue. Enhancements currently in progress include direct and very fast execution for a substantial subset of the PVS language (this supports computational reflection [48], as well as improved validation of specifications [18]), and faster and more automated theorem proving. Those planned for the near future include support for refinement and a more open system architecture.

References

1. Parosh Aziz Abdulla, Aurore Annichini, Saddek Bensalem, Ahmed Bouajjani, Peter Habermehl, and Yassine Lakhnech. Verification of infinite-state systems by combining abstraction and reachability analysis. In CAV'99 [8].
2. Andren Alborghetti, Angelo Gargantini, and Angelo Morzenti. Providing automated support to deductive analysis of time critical systems. In Mehdi Jazayeri and Helmut Schauer, editors, *Software Engineering–ESEC/FSE '97: Sixth European Software Engineering Conference and Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1301 of *Lecture Notes in Computer Science*, pages 211–226, Zurich, Switzerland, September 1997. Springer-Verlag.
3. Rajeev Alur and Thomas A. Henzinger, editors. *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, NJ, July/August 1996. Springer-Verlag.
4. Myla Archer and Constance Heitmeyer. Mechanical verification of timed automata: A case study. In *IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, pages 192–203, Brookline, MA, June 1996. IEEE Computer Society.

5. Saddek Bensalem, Yassine Lakhnech, and Hassen Saïdi. Powerful techniques for the automatic generation of invariants. In Alur and Henzinger [3], pages 323–335.
6. Bettina Buth. PAMELA + PVS. In Michael Johnson, editor, *Algebraic Methodology and Software Technology, AMAST'97*, volume 1349 of *Lecture Notes in Computer Science*, pages 560–562, Sydney, Australia, December 1997. Springer-Verlag.
7. Ricky W. Butler and Jon A. Sjogren. A PVS graph theory library. NASA Technical Memorandum 1998-206923, NASA Langley Research Center, Hampton, VA, February 1998.
8. *Computer-Aided Verification, CAV '99*, Lecture Notes in Computer Science, Trento, Italy, July 1999. Springer-Verlag. To appear.
9. Raphaël Couturier and Dominique Méry. An experiment in parallelizing an application using formal methods. In Hu and Vardi [25], pages 345–356.
10. Judith Crow and Ben L. Di Vito. Formalizing Space Shuttle software requirements: Four case studies. *ACM Transactions on Software Engineering and Methodology*, 7(3):296–332, July 1998.
11. Drew Dean. Static typing with dynamic linking. In *Fourth ACM Conference on Computer and Communications Security*, pages 18–27, Zurich, Switzerland, April 1997. Association for Computing Machinery.
12. Darryl Dieckman, Perry Alexander, and Philip A. Wilsey. ActiveSPEC: A framework for the specification and verification of active network services and security policies. In Nevin Heintze and Jeannette Wing, editors, *Workshop on Formal Methods and Security Protocols*, Indianapolis, IN, June 1998. Informal proceedings available at `http://www.cs.bell-labs.com/who/nch/fmsp/program.html`.
13. Bruno Dutertre and Victoria Stavridou. Formal requirements analysis of an avionics control system. *IEEE Transactions on Software Engineering*, 23(5):267–278, May 1997.
14. Steve Easterbrook, Robyn Lutz, Richard Covington, John Kelly, Yoko Ampo, and David Hamilton. Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering*, 24(1):4–14, January 1998.
15. *Formal Methods Europe FME '97*, volume 1313 of *Lecture Notes in Computer Science*, Graz, Austria, September 1997. Springer-Verlag.
16. Simon Fowler and Andy Wellings. Formal development of a real-time kernel. In *Real Time Systems Symposium*, pages 220–229, San Francisco, CA, December 1997. IEEE Computer Society.
17. Ganesh Gopalakrishnan and Phillip Windley, editors. *Formal Methods in Computer-Aided Design (FMCAD '98)*, volume 1522 of *Lecture Notes in Computer Science*, Palo Alto, CA, November 1998. Springer-Verlag.
18. David Greve. Symbolic simulation of the JEM1 microprocessor. In Gopalakrishnan and Windley [17], pages 321–333.
19. David Hardin, Matthew Wilding, and David Greve. Transforming the theorem-prover into a digital design tool: From concept car to off-road vehicle. In Hu and Vardi [25], pages 39–44.
20. Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681, Oxford, UK, March 1996. Springer-Verlag.
21. Mats P. E. Heimdahl and Barbara J. Czerny. Using PVS to analyze hierarchical state-based requirements for completeness and consistency. In *IEEE High-Assurance Systems Engineering Workshop (HASE '96)*, pages 252–262, Niagara on the Lake, Canada, October 1996.

22. John Hoffman and Charlie Payne. A formal experience at Secure Computing Corporation. In Hu and Vardi [25], pages 49–56.
23. Jozef Hooman. Compositional verification of real-time applications. In Willem Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference (Revised lectures from International Symposium COMPOS'97)*, volume 1536 of *Lecture Notes in Computer Science*, pages 276–300, Bad Malente, Germany, September 1997. Springer-Verlag.
24. Ravi Hosabettu, Mandayam Srivas, and Ganesh Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In CAV'99 [8].
25. Alan J. Hu and Moshe Y. Vardi, editors. *Computer-Aided Verification, CAV '98*, volume 1427 of*Lecture Notes in Computer Science*, Vancouver, Canada, June 1998. Springer-Verlag.
26. Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrick Tews. Reasoning about Java classes. In *Proceedings, ObjectOriented Programming Systems, Languages and Applications (OOPSLA'98)*, pages 329–340, Vancouver, Canada, October 1998. Association for Computing Machinery. Proceedings issued as ACM SIGPLAN Notices Vol. 33, No. 10, October 1998.
27. Pertti Kellomäki. Verification of reactive systems using DisCo and PVS. In FME [15], pages 589–604.
28. Paul S. Miner and James F. Leathrum, Jr. Verification of IEEE compliant subtractive division algorithms. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 64–78, Palo Alto, CA, November 1996. Springer-Verlag.
29. Abdel Mokkedem, Ravi Hosabettu, and Ganesh Gopalakrishnan. Formalization and proof of a solution to the PCI 2.1 bus transaction ordering problem. In Gopalakrishnan and Windley [17], pages 237–254.
30. César Muñoz. PBS: Support for the B-method in PVS. Technical Report SRI-CSL 99-1, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1999.
31. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Alur and Henzinger [3], pages 411–414.
32. Sam Owre, John Rushby, and N. Shankar. Integration in PVS: Tables, types, and model checking. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 366–383, Enschede, The Netherlands, April 1997. Springer-Verlag.
33. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
34. Seungjoon Park and David L. Dill. Verification of cache coherence protocols by aggregation of distributed transactions. *Theory of Computing Systems*, 31(4):355–376, 1998.
35. N.S. Pendharkar and K. Gopinath. Formal verification of an O.S. submodule. In V. Arvind and R. Ramanujin, editors, *18th Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 1530 of *Lecture Notes in Computer Science*, pages 197–208, Madras, India, December 1998. Springer-Verlag.
36. Holger Pfeifer, Detlef Schwier, and Friedrich W. von Henke. Formal verification for time-triggered clock synchronization. In Rushby [41], pages 193–212.

37. Amir Pnueli and Tamara Arons. Verification of data-insensitive circuits: An in order-retirement case study. In Gopalakrishnan and Windley [17], pages 351–368.
38. S. P. Rajan, M. Fujita, K. Yuan, and M. T-C. Lee. ATM switch design by high level modeling, formal verification, and high level synthesis. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 3(4), October 1998.
39. John Rushby. PVS bibliography. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA. Constantly updated; available at `http://www.csl.sri.com/pvs-bib.html`.
40. John Rushby. Formal methods and their role in the certification of critical systems. In Roger Shaw, editor, *Safety and Reliability of Software Based Systems (Twelfth Annual CSR Workshop)*, pages 1–42, Bruges, Belgium, September 1995. Springer.
41. John Rushby, editor. *Dependable Computing for Critical Applications–7*, Dependable Computing and Fault Tolerant Systems, San Jose, CA, January 1999. IEEE Computer Society. To appear (page numbers refer to preliminary proceedings).
42. John Rushby, Sam Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.
43. Hassen Saïdi and N. Shankar. Abstract and model check while you prove. In CAV'99 [8].
44. Jens U. Skakkebjk and N. Shankar. Towards a Duration Calculus proof assistant in PVS. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 660–679, Lübeck, Germany, September 1994. Springer Verlag.
45. David W. J. Stringer-Calvert. Mechanical Verification of Compiler Correctness. PhD thesis, University of York, Department of Computer Science, York, England, March 1998. Available at `http://www.csl.sri.com/ dave.sc/papers/thesis.html`.
46. Kothanda Umamageswaran, Krishnan Subramani, Philip A. Wilsey, and Perry Alexander. Formal verification and empirical analysis of rollback relaxation. *Journal of Systems Architecture (formerly published as Microprocessing and Microprogramming: the Euromicro Journal)*, 44:473–495, 1998.
47. Ben L. Di Vito. A model of cooperative noninterference for integrated modular avionics. In Rushby [41], pages 251–268.
48. Friedrich von Henke, Stephan Pfab, Holger Pfeifer, and Harald Rueß. Case studies in meta-level theorem proving. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs '98*, volume 1479 of Lecture Notes in Computer Science, pages 461–478, Canberra, Australia, September 1998. Springer-Verlag.
49. M. Wahab. Verification and abstraction of flow-graph programs with pointers and computed jumps. Research Report CS-RR-354, Department of Computer Science, University of Warwick, Coventry, UK, November 1998. Available at `http://www.dcs.warwick.ac.uk/pub/reports/rr/354.html`.
50. Matthew M. Wilding, David S. Hardin, and David A. Greve. Invariant performance: A statement of task isolation useful for embedded application integration. In Rushby [41], pages 269–282.

The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

Overview over the Project

Oscar Slotosch

Institut für Informatik, Technische Universität München
80290 München, Germany

Abstract. The software development project Quest provides tools to build correct software, especially for embedded systems. It connects the CASE-tool AUTOFOCUS to the formal development tool VSE II, and the model checker SMV. To increase quality of non-formally developed programs a test environment is realized, containing a connection to the test-case classification tool CTE.
Proving the correctness of the emergency closing system of a storm surge barrier is a running example within the project Quest.

1 Structure of Quest

The project Quest is carried out for the German "Bundesamt für Sicherheit in der Informationstechnik" (BSI). One of the main tasks of the BSI is to increase the assurance and trustworthiness of security and safety critical systems. Therefore the aim of the project Quest is to enrich the practical software development process by the coupling existing methods and tools using formal techniques in order to ensure the correctness of critical parts of systems. For the development of large systems a scalable concept is required. To achieve this goal we choose hierarchical, graphical description techniques (supported by CASE-tools) and a compositional logic with theorem prover support. For small systems model checkers can be used to verify critical properties automatically.

The combination with traditional software engineering methods is achieved by specification-based test methods to generate test-cases for non-critical components. The tools developed within Quest translate the graphical concepts into a logical formalism and systematically generate test-cases. To support an integrated, iterative development process a graphical visualization of some formulas is planned by retranslating them to the CASE-tool.

The main phases of the project Quest are: the analysis phase, where the tools AUTOFOCUS (CASE-tool), VSE II (theorem prover), SMV (model checker) and CTE (test case assistant) have been selected, the design phases, where the translation concepts have been fixed, and the implementation phase where the translators are realized and integrated into the tools. Quest started out at 1.10.1997 and will end at 30.9.1999. The project Quest and also this paper are structured into four parts: the connection from AUTOFOCUS to VSE II, the connection from

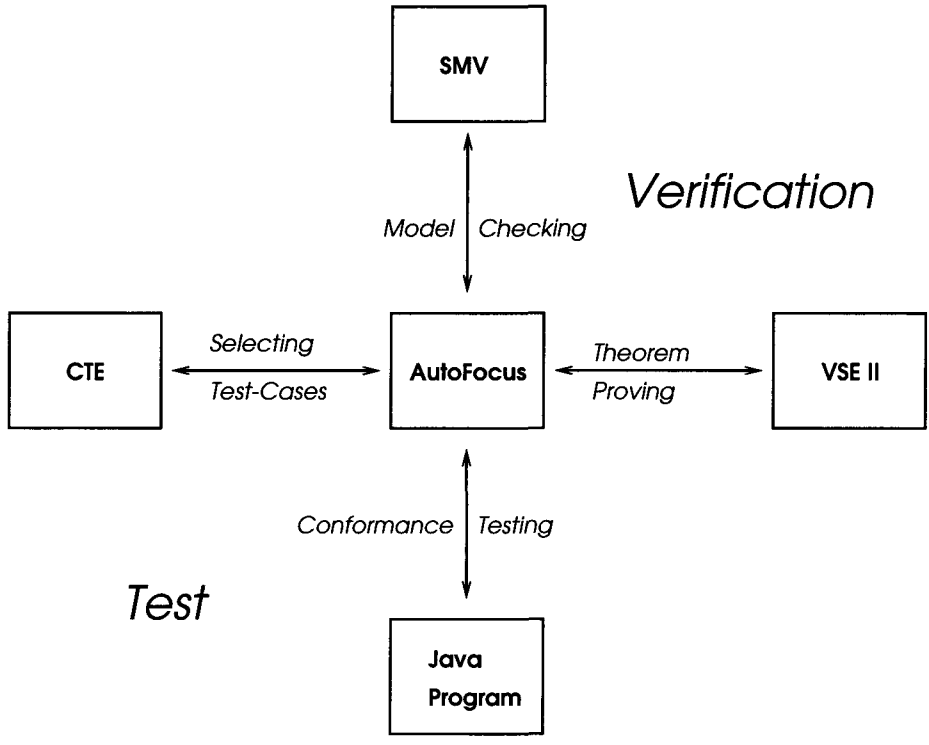


Fig. 1. Tool Structure of Quest

AUTOFOCUS to SMV, the test environment and the case study. An integration of the Quest tools into the software development process using the V-model is described in [BS99].

Figure 1 shows the structure of the tools of the project Quest. The boxes represent the selected tools, and the arrows between them denote the connections that are designed and implemented within the project Quest. We integrate the tools by defining interfaces and (sometimes partial) translations between the used concepts. The interfaces could also be used by other tools. This paper shortly describes the connections between the tools, and the case study that is developed using tools.

The project is carried out at the Technical University of Munich (Group of Prof. Broy), together with the DFKI (German Research Center for AI), Daimler Benz Research, and ist (innovative software technologie). The following persons are contributing with the author to the success of Quest: F. Koob, M. Ullmann, S. Wittmann (BSI), W. Stephan, A. Wolpers, G. Rock, H. Mantel (DFKI), P. Baur, T. Plasa, M. Popall (ist), T. Klein, S. Sadeghipour (DB), M. Broy, S. Merz, J. Philipps, O. Müller, I. Krüger, H. Lötzbeyer, P. Braun, R. Sandner, P. Gierl, T. Sprenger, G. Wimmel, and D. Chibisov (TUM).

2 The Connection between AUTOFOCUS and VSE II

The CASE-tool AUTOFOCUS [HMR+98,HMS+98] has been designed for the development of correct embedded systems. It bases on the formal concepts of FOCUS [BDD+92]. AUTOFOCUS offers graphical description techniques for structure, behaviour, and interaction. The descriptions use functional data types, and allow pattern matching in the definition of functions and in the transitions of the behavioural view. All views can be structured hierarchically, and consistency of the descriptions can be checked. AUTOFOCUS has a synchronous simulation[1] to validate the specifications by rapid prototyping. AUTOFOCUS is extended within the project Quest to store properties of components.

The VSE II system [RSW97] is an extension of the VSE [HLS+96] system by concepts of TLA, hence it has asynchronous, action oriented semantics. VSE II uses, like AUTOFOCUS, functional data types. To define correct translations from AUTOFOCUS components to VSE II a scheduler is introduced within VSE II to enforce synchronization of the components. Since AUTOFOCUS components can be hierarchic, the translation introduces one scheduler for every level of abstraction. The communication channels between components are modelled by combining the input and output channels of the components. This translation is also applied to properties of specifications that are translated to VSE II.

There is also a partial retranslation from VSE II to AUTOFOCUS. It inverts the above translation and can be used to visualize structures of VSE II specifications. Furthermore changes made within VSE II can be incorporated into the original AUTOFOCUS model of the system.

This connection between the CASE-tool AUTOFOCUS and VSE II supports a user-friendly description of systems and allows to prove arbitrary properties using the general theorem prover of VSE II.

3 The Connection between AUTOFOCUS and SMV

We chose the SMV system [McM92] because is offers an efficient checking procedure for synchronous models. Therefore no scheduler is required. The translation to SMV generates a module for every component and combines the modules using variables for message channels. This translation is quite natural, however since SMV has neither functions (except boolean and integer arithmetic) nor functional data types integrated it is necessary to eliminate these data types during the translation from AUTOFOCUS to SMV. Of course the static elimination works only for finite data types.

The retranslation from SMV to AUTOFOCUS, depends on the model checking result. In the case the property is valid, this information is passed to AUTOFOCUS. In the case SMV produces a counterexample, this is translated into an

[1] The simple (not object-oriented) component semantics of AUTOFOCUS and the synchronous execution model characterize the main application domain of AUTOFOCUS: embedded systems.

event trace that shows the interaction between the components that leads to a contradiction of the desired property.

Furthermore abstraction techniques are integrated to check properties of large systems by reducing systems to simpler ones that can be checked. The correctness of this simplifications has to be proved (for every property, and every abstraction) using the VSE II system.

4 Test Environment

The test environment is part of the project Quest, because testing is important to increase quality of software, especially if the development is not completely formal. Extended Event Traces (EETs) are a graphical notation of AUTOFOCUS to visualize interactions of components with other components and the environment. EETs are used to represent test-cases.

The test environment computes a "transition tour" i.e. a minimal sequence of transitions that covers all possible transitions of the state transition diagram. If the behaviour description contains no variables, the input/output sequences for the test run can be generated automatically. Otherwise the user has to select appropriate input values for the variables of the transitions to construct the input/output sequences for the test run. The CTE tool [GWG95] supports the user in managing classifications of input variables, basing on the types of the variables.

For the execution of test runs a driver is implemented that passes the values of the EET to the Java program under test, and compares the results with the output values of the EET.

The test environment of Quest supports a systematic, specification-based test generation and execution. Furthermore a graphical notation for test-cases is given.

5 Case Study: Emergency Closing System

To demonstrate the benefits of Quest for the practical development of safety critical systems, a real-world case study has been selected. The case study is carried out together with the Dutch SIMTECH company, that is currently designing the system. For this project a high level of integrity is required that can only be reached by a formal correctness proof.

The task is to correctly develop a realization for the emerging closing system of the storm surge barrier in the Eastern Scheld, that prevents the Netherlands from floodings like 1953. The system has six sensors for the inside and outside water levels and determines if the barrier has to be closed, and if it can be reopened. The correctness of the open- and close-signals has to be verified formally.

The informal specification is textual with some graphical descriptions of the systems structure and behaviour. The current state of the case study is that a

complete formal model has been built using AUTOFOCUS and the translation into the SMV systems resulted into a checkable system. Verification and the generation of tests is ongoing work.

6 Conclusion

The project Quest integrates formal methods and practical software development into a framework, that allows to develop correct software for embedded systems. Instead of general software development (like UML), we focus on embedded systems, because the formal foundations for this application domain are ready for practical applications.

Within the Quest framework it will be possible to develop correct software and to certify a high level of integrity for the developed systems.

For comments on previous versions of this paper we thank Heiko Lötzbeyer, Robert Sandner, and Stefan Wittmann.

References

[BDD+92] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T.F. Gritzner, and R. Weber. The design of distributed systems - an introduction to FOCUS. Technical Report TUM-I9203, Technische Universität München, Januar 1992.

[BS99] M. Broy and O. Slotosch. Enriching the Software Development Process by Formal Methods. 1999. this volume.

[GWG95] M. Grochtmann, J. Wegner, and K. Grimm. Test Case Design Using Classification Trees and the Classification-Tree Editor. In *Proceedings of 8th International Quality Week, San Francisco*, pages Paper 4–A–4, May 30-June 2 1995.

[HLS+96] D. Hutter, B. Langenstein, C. Sengler, J. Siekmann, W. Stephan, and A. Wolpers. Deduction in the Verification Support Environment(VSE). In Marie-Claude Gaudel and James Woodcock, editors, *Proceedings Formal Methods Europe 1996: Industrial Benefits and Advances in Formal Methods.* Springer-Verlag, 1996.

[HMR+98] F. Huber, S. Molterer, A. Rausch, B. Schätz, M. Sihling, and O. Slotosch. Tool supported Specification and Simulation of Distributed Systems. In B. Krämer, N. Uchihira, P. Croll, and S. Russo, editors, *Proceedings International Symposium on Software Engineering for Parallel and Distributed Systems, pp. 155-164, ISBN 0-8186-8467-4*, pages 155–164. IEEE Computer Society, Los Alamitos, California, 1998.

[HMS+98] F. Huber, S. Molterer, B. Schätz, O. Slotosch, and A. Vilbig. Traffic Lights - An AUTOFOCUS Case Study. In *1998 International Conference on Application of Concurrency to System Design*, pages 282–294. IEEE Computer Society, 1998.

[McM92] K.L. McMillan. The SMV system, Symbolic Model Checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[RSW97] G. Rock, W. Stephan, and A. Wolpers. Tool Support for the Compositional Development of Distributed Systems. In *Proc. Formale Beschreibungstechniken für verteilte Systeme,GI/ITG-Fachgespräch.* GMD-Studien Nr. 315, ISBN: 3-88457-514-2, 1997.

VSE: Controlling the Complexity in Formal Software Developments

Dieter Hutter, Heiko Mantel, Georg Rock, Werner Stephan, Andreas Wolpers[1], Michael Balser, Wolfgang Reif, Gerhard Schellhorn, and Kurt Stenzel[2]

[1] German Research Center for Artificial Intelligence (DFKI),Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany, email: <name>@dfki.de
[2] Abt. Programmiermethodik, Universität Ulm, D-89069 Ulm, Germany, email:<name>@informatik.uni-ulm.de

Abstract. We give an overview of the enhanced VSE system which is a tool to formally specify and verify systems. It provides means for structuring specifications and it supports the development process from the specification of a system to the code generation. Formal developments following this method are stored and maintained in an administration system that guides the user and maintains a consistent state. An integrated deduction system provides proof support for the deduction problems arising during the development process.

1 Introduction

The reliability of complex software systems is becoming increasingly important for technical systems. Malfunctioning of software systems caused by design flaws or faulty implementations may lead to loss or garbling of data, breach of security, danger to life and limb, and, in almost all cases severe economic losses.

In order to allow for an industrial development of software according to the highest IT security criteria (ITSEC), the VSE tool [5] was developed on behalf of the german agency BSI from 1991 to 1994. It aimed at a thorough support in all phases of a *formal development methodology.* This includes in particular an adequate management of software developments, an efficient proof support, and a sophisticated user interface. After a period of evaluation in various large applications (e.g. a booking system for radio transmissions, an access control system for nuclear power plants, and a security model for digital signatures), VSE [5] is now applied in a commercial formal development.

Based on these experiences, the VSE tool is now enlarged and extended with respect to comprehensive methods in order to deal with *distributed* and *concurrent* systems [9] and with respect to an even more efficient and uniform proof support which makes use of the implicit structuring of the arising proof obligations. A refined *correctness management* allows for an evolutionary software development without proving everything from scratch again. Again the development of VSE-II is accompanied by case studies, where for instance the safety critical part of a robot control system working in a fusion reactor is re-developed [8]. We have formalized a smaller case study in VSE-II, presented in the next section, as a running example in this paper.

2 Example Specification

We specify a communication of two components via a buffered FIFO-channel with a finite capacity (see Figure 1). The producer creates arbitrary values (using the internal variable **x**), and sends them to the channel component along **in**. If the capacity **max** of the internal buffer **q** of the channel is not exceeded, the channel receives these values and stores them into the buffer. The channel eventually transmits received values according to the FIFO principle along **out** to the consumer. The consumer receives these values, stores them in **y**, and uses them in some computation specified elsewhere.

As illustrated in Figure 1, producer, channel, and consumer work concurrently and are specified as separate entities using pseudo code. The components

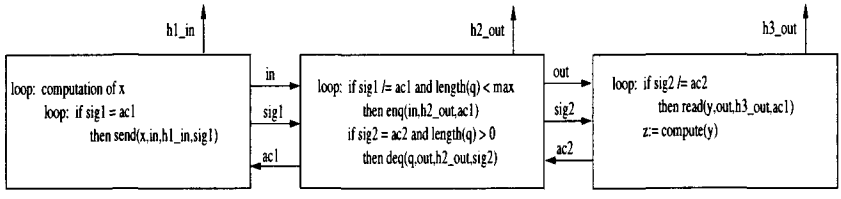


Fig. 1. Producer–Channel–Consumer

exchange data in a classical way of handshaking: A sender informs a receiver that a value has been transmitted by a *signal* (using **sig1** or **sig2**). A receiver informs a sender that a value has been received by an *acknowledge* (using **ac1** or **ac2**). **send(x,in,h1_in,sig1)** denotes an atomic action which writes the value of **x** to **in**, updates the history variable **h1_in**, and signals along **sig1**. Similarly **enq**, **deq**, and **read** denote atomic actions. The variables **h2_out**, and **h3_out** are history variables which store the former values of **out**.

As we have to verify, the protocol implemented by the communication through these channels will guarantee that no values are lost. Thus, we formulate an invariant property for the communication between the producer and the consumer as $\exists l.h1_in = l \circ h2_out$. Intuitively, this states that the history of the values sent by the channel is always a postfix of the history of the values sent by the producer.

3 Development Graph

VSE is based on a methodology to use the structure of a given specification (e.g. parameterisation, actualisation, enrichment, or modules) to distribute also the deductive reasoning into local theories. Each theory is considered as an encapsulated unit, which consists of its own deductive system and its local theory. Relations between different theories, as they are given by the model-theoretic structure of the specification, are represented by different links between theories.

Each theory maintains its own set of consequences or lemmata obtained by using local axioms and other formulas included from linked theories.

This method of a structured specification *and* verification is reflected in the central data structure of a *development graph* (see Figure 2), the nodes of which correspond to the units mentioned above. It also provides a graphical interface for the system under development. Different types of specifications are displayed as different types of nodes, e.g. abstract data types as hexagons, while the relations between the nodes are displayed as links in the development graph.

Lists (*Lists*) and queues (*Queues*) are generic specifications parameterised with specification *Elems*. *Channel_Data_LQs* unites actualisations of lists and queues, the actual parameter being *Channel_Datas*. The specifications of lists and queues are hierarchically structured; the basic specifications *Basic_Lists* and *Basis_Queues* contain freely generated types for which axioms (and executable code) are generated automatically. These basic specifications are enriched by additional functions and predicates. *Color* and *Label* are special abstract specifications which define enumeration types. A set of predefined specifications—*Naturals* and *Boolean* in the example—is also provided.

State machines are specified with the help of temporal logic constructs (see section 5). The state variables take values from the abstract datatypes used by the state machine: the *channel* uses the theory *Queues* for its internal buffer. State machines are shown as octagons in the development graph, e.g. *Producer*, *Channel* or *Consumer* in Figure 2. These three state machines are composed to the state machine *Combined_Producer_Consumer* by identifying the input and output variables of each component, and hiding the internal communication variables.

The *Safety_Model* contains properties which have to be satisfied by *Combined_Producer_Consumer*.

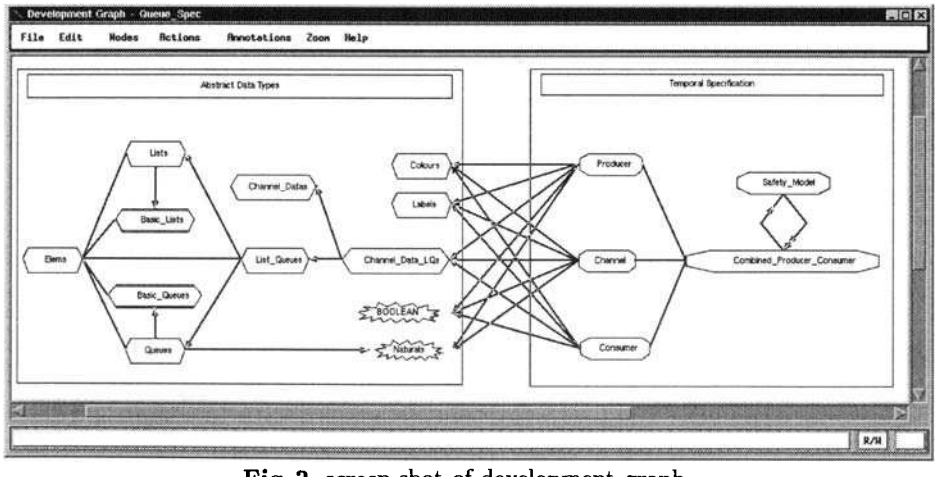


Fig. 2. screen-shot of development graph

4 Functional Modelling

VSE supports the development of functional systems using algebraic specifications. Elementary algebraic specifications contain sorts, functions, and predicates; the intended semantics of the operations is defined by axioms in full first order predicate logic. Additional language constructs allow for the formulation of generation principles to define inductive data types. For frequently used "basic" specifications (freely generated data types), axioms and ADA or C code are automatically derived. Elementary algebraic specifications can be combined using the structuring operations non-disjoint union, enrichment, and actualisation of specifications. These structuring operations allow the independent (modular) refinement of algebraic specifications [12].

```
THEORY Lists
 PURPOSE "parameterised list type"
 PARAMS Elems
 USING Basic_Lists[elems]
 FUNCTIONS  append   : list, list -> list
 PREDICATES is_empty : list
 VARS l, l1, l2 : list
 AXIOMS
    FOR append : DEFFUNC append(l1, l2) =
                 IF l1 = nil THEN l2
                 ELSE cons(first(l1), append(rest(l1), l2)) FI
    FOR is_empty : is_empty(l) <-> l = nil
THEORYEND
```

Fig. 3. example algebraic specification of lists

In Figure 3 the specification of lists is given as an example. The basic specification *Basic_Lists* is used and additional operations *append* and *is_empty* are defined. Axioms for the operators can be written as arbitrary first order formulas or in an algorithmic style (*DEFFUNC*), the latter enabling the deduction unit to automatically derive fitting induction orders.

Refining algebraic specifications in VSE is done by defining modules which contain functions and procedures of an imperative programming language. These functions and procedures implement the operations of an export relative to an import specification. The correctness of a refinement is guaranteed, if the automatically generated proof obligations can be shown. One set of proof obligations is concerned with termination of each function and procedure, another set demands conformity with the axioms of the export specification. The proof obligations are expressed in Dynamic Logic [10], [11]; main tactics for proving the obligations are symbolic execution and induction. The proof component uses heuristics to highly automate the deduction process.

As the operations of the import specification of a module are usually not directly available in any machine or programming language, the programs are sometimes called abstract programs. At the end of the development process, the entire refinement structure is automatically translated into ADA or C code. Not

every operation of the specification has to be formally refined. Instead, formal development of programs can be integrated with conventional development by mapping operations onto conventionally designed programs. Implementations for predefined data types (naturals, sets, arrays, etc.) are provided.

5 Specifications of Concurrent Systems

5.1 Elementary Specifications

For the specification of state transition systems (see Producer, Channel (see Figure 4) and Consumer in section 3), a specification language close to TLA [1, 7, 2] is used. In addition to the theory of compositional development presented in [2], which covers the composition of systems using input and output variables, *shared variables* are supported by the structuring operators in VSE-II (see section 5.2). The form of a specification of a component, discussed in [2], is

$$\exists\, x_1, \ldots, x_n.(\text{INIT} \wedge \Box[\text{POSSIBLE-STEPS}]_{\bar{v}} \wedge \text{FAIR}),$$

where POSSIBLE-STEPS are the steps made by the system, $\bar{v}$ is the stuttering index, which contains (a part of) the (flexible) variables of the system, and FAIR stands for the fairness requirements of the system.

5.2 Structuring of Specifications

In VSE-II we provide two operators to structure state-based specifications: **combine** and **include**. In this paper we focus on the **combine**-operator which models the concurrent execution of two components. As can be seen in Figure 2, the Combined_Producer_Consumer state machine consists of the composition of the state machines Producer, Channel (shown in Figure 4), and Consumer.

Concurrency is modeled by considering all possible *interleavings* of actions of the combined systems. Basically a behavior σ, which represents a sequence of states of the specified system, is a behavior of the combined system if and only if it is a behavior of every component of the system. However, in order to model the concurrent execution of, say $\mathcal{S}_1$ and $\mathcal{S}_2$, by conjunction, we have to allow environment steps in the (local) specifications of $\mathcal{S}_1$ and $\mathcal{S}_2$. In [2] environment steps are modeled by stuttering. This technique only works for communication by input-output variables, not in connection with shared variables. A more general approach is to associate a "color" with each component and to mark each step in a behavior by the color of the component which has done the step. The (canonical) specification is then changed to:

$$\Box(((\text{POSSIBLE} - \text{STEPS}) \wedge \mathit{color} = \text{mycolor}\ ) \vee \mathit{color} \neq \text{mycolor}\ ),$$

where *color* is a variable and mycolor is a constant. The colors itself are hidden after the composition and not externally visible. So the external behavior remains the same.

```
TLSPEC Channel
 PURPOSE
 "Channel for data transmission from producer to consumer"
 USING BOOLEAN;NATURALS;Colours;Labels;Channel_Data_LQs
 DATA
      IN sig1,ac2 : bool
      IN in : channel_data
      OUT ac1,sig2 : bool
      OUT out : channel_data
      OUT h2_out : list
      INTERNAL at : label
      INTERNAL q : queue
 ACTIONS
 Init_Channel ::= ac1 = T AND sig2 = T AND  q = emptyqueue AND
                  h2_out = emptylist
 Channel_Action3 ::= out' = out AND sig2' = sig2 AND h2_out' = h2_out AND
                     ac1' = ~ ac1 AND q' = enqueue(q, inn)
 ...
 /* Assumptions made by the channel about the producer. */
 ASSUME ... (sig1 /= ac1 -> inn' = inn) ...
 SPEC INITIAL Init_Channel
      TRANSITIONS [..., Channel_Action3, ...] {q, sig2, out, h2_out, ac1}
 HIDE q
TLSPECEND
```

Fig. 4. A part of the channel specification from Figure 1.

As mentioned before communication can be modeled by in/output variables as well as by shared variables. This communication relies on the fact that several systems can read or write the same variable(s).

The second operator for the composition of systems, say $\mathcal{S}_1$ and $\mathcal{S}_2$, is **include**. $\mathcal{S}_1$ **include** $\mathcal{S}_2$, represents a mechanism to provide certain functionalities which are often used in the specification of systems without re-specifying them. It allows furthermore for a hierarchical composition of state machines. The semantics of $\mathcal{S}_1$ **include** $\mathcal{S}_2$ is $\mathcal{S}_1 \wedge \mathcal{S}_2$. The steps of $\mathcal{S}_2$ in the composed system are not left open but the actions of the system $\mathcal{S}_2$ occur now in the system $\mathcal{S}_1$. In this way the externally visible behavior of $\mathcal{S}_1$ is also determined by the system $\mathcal{S}_2$. This composition style can end up in a inconsistent system. So proof obligations arise which have to be proved in order to assure the consistency of the composition.

6 Proof Engineering

In order to tackle arising proof obligations there is a need for a *structured deduction* decomposing large proof obligations into simpler tasks and synthesizing an overall proof from the arising partial solutions. This enables the use of specialized methods to solve specific problems and also eases the speculation of lemmata needed to prove the theorem. Furthermore, the use of proof strategies

(like e.g. [11, 9]) based on specialized logics like Dynamic Logic [10] for sequential programs or TLA [7] for reactive and concurrent systems allows for an adequate representation of specifications and arising proof problems.

In VSE the knowledge about how to tackle specific proof situations is encoded into a bundle of individual tactics. The accumulation of various tactics imposes an emerging functionality which is able to prove complex theorems in a goal directed way. All these tactics operate on a common representation of the actual proof state which is reflected in a proof tree annotated by additional tactical knowledge. This proof tree is visualized to allow the user to give strategic advise following the *direct manipulation paradigm*. Tactics may prune or refine this proof tree and represent the *algorithmic* part of the proof search. In VSE proof decisions can be withdrawn by backtracking steps chronologically as well as by pruning arbitrary branches of the proof tree. The approach combines a high degree of automation with an elaborate interactive proof engineering environment. VSE uses annotated terms [4] as a framework to maintain various planning information during an actual proof search. They provide a simple mechanism to propagate knowledge arising within the proof search. Within this framework tactics are able to communicate their achievements to following tactics and the progress of the proof is represented by a syntactical representation of differences between the actual state and possible goal states.

Consider for instance our example illustrated in Figure 2. In order to verify that all data sent by the producer will be either stored in the local queue of the channel or will be read by the consumer we have to prove that specific properties of these components remain invariant. This obligation is decomposed into a proof that the initialization guarantees the invariant and that each action will preserve the invariant. The latter can be formalized as follows: $Action(\overline{X}, \overline{X'}) \rightarrow (Inv(\overline{X}) \rightarrow Inv(\overline{X'}))$. In order to tackle such a proof obligation we use a proof technique which is divided into two steps: First, appropriate rewrite rules are synthesized from $Action(\overline{X}, \overline{X'})$, the right hand sides of which do not contain any primed variables and allow us to remove occurrences of primed variables in $Inv(\overline{X'})$. Second, $Inv(\overline{X'})$ is analyzed for occurrences of primed variables guiding the appropriate selection of the computed rewrite rules. This gives rise to an elaborated case analysis in order to deal with the conditionals of the rewrite rule. Proving the individual cases we distinguish two cases: In the first case where the selected rewrite rules contain non-primed variables in their value-part, we try to reduce $Inv(\overline{X'})$ to $Inv(\overline{X})$ which is done with the help of annotated terms [4] allowing for an explicit representation of the differences between both. In the second case, where no unprimed variables are introduced by applying the rewrite rules, the proof has to be done from scratch without using $Inv(\overline{X'})$.

Often the attempt to prove a lemma or proof obligation fails and detected errors in the specification or implementation have to be corrected. In VSE an elaborate correctness management supervises the application of corrections and invalidates only those proofs which really are affected by the modifications. Furthermore, invalid proofs can be automatically reused [13].

7 Conclusion

VSE enables large software projects to be formally developed. It supports the whole development process from system specification to executable code. Functional systems as well as state based reactive concurrent systems can be specified. Systems are designed using an elaborate graphical environment. The structure of a system is displayed as a development graph. The structuring operations ensure that specifications can be separately refined and properties can be proved locally to its components. Powerful proof support for arising proof obligations is provided. An elaborate correctness management and automatic proof reuse simplify the iterative process of (failed) proof attempts. Altogether VSE is a tool which makes it possible to use formal methods for the development of high assurance systems in an industrial context.

References

1. M. Abadi and L. Lamport: The existence of refinement mappings. *Theoretical Computer Science, North Holland, Elsevier Science Publishers B.V.*, 82(2):253–284, May 1991.
2. M. Abadi and L. Lamport: Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.
3. D. Hutter: Hierarchical proof planning using abstractions. *10th FLAIRS'97*, Daytona Beach, Florida, 1997
4. D. Hutter: Colouring Terms to control equational reasoning. *Journal of Automated Reasoning*, Kluwer-Publishers, Vol. 18, pp. 399-442, 1997
5. D. Hutter, B. Langenstein, C. Sengler, J. H. Siekmann, W. Stephan, A. Wolpers: Deduction in the Verification Support Environment (VSE). In Marie-Claude Gaudel and James Woodcock, editors, *Proceedings Formal Methods Europe 1996: Industrial Benefits and Advances in Formal Methods*. Springer-Verlag, Berlin, Germany, 1996.
6. IT-Sicherheitskriterien. Bundesanzeiger, 1989.
7. L. Lamport: The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3), 1994.
8. G. Rock, W. Stephan, and A. Wolpers: Assumption–commitment specifications and safety-critical systems. In *Tagungsband 8. GI/ITG-Fachgespräch Formale Beschreibungstechniken für verteilte Systeme*, 1998. to appear.
9. G. Rock, W. Stephan, and A. Wolpers: Modular reasoning about structured TLA specifications. In *Proceedings TOOLS'98*, 1998. to appear.
10. R. Goldblatt: Axiomatising the logic of computer programming. Springer LNCS 130, 1982.
11. M. Heisel, W. Reif, and W. Stephan: A Dynamic Logic for program verification. In A. Meyer and M. Taitslin, editors, *Logical Foundations of Computer Science*. Springer LNCS 363, 1989.
12. W. Reif. Correctness of generic modules. In Nerode and Taitslin, editors, *Symposium on Logical Foundations of Computer Science*. Springer LNCS 620, 1992.
13. W. Reif and K. Stenzel. Reuse of proofs in software verification. In R. Shyamasundar, editor, *Foundation of Software Technology and Theoretical Computer Science. Proceedings*. Springer LNCS 761, 1993.

The wHOLe System

Mark E Woodcock

Department of Defense, Fort Meade MD USA
Department of Computer Science and Electrical Engineering, UMBC, USA
woodcock@cs.umbc.edu

Abstract. A theorem-proving system derived from Higher Order Logic (HOL) is described. The system is designed to support the verification of code written in its own implementation language (SML/NJ), and to allow code which is proven to preserve equality of hol_terms to be used to extend the system. The extension is shown to be theoretically conservative, and the implementation to be reliable. It is hoped that significant efficiency gains can be realized employing this technique (Computational Reflection).

1 SYSTEM SOUNDNESS

1.1 Impossible Ideal

The ideal of formal logic correctness is known as Hilbert's programme. Hilbert hoped to devise a logical engine which could: 1) solve any logical problem, and 2) be used to demonstrate the soundness of the engine itself. Goedel's incompleteness theorems proved that the programme was unobtainable in both respects. The theorems established that:

1) for any interesting logic (i.e. one that includes at least arithmetic over the integers) there are statements in the logic that are true, but can not be proved

2) no formal logic is sufficiently powerful to establish its own soundness.

This induces a temptation to justify the soundness of a desired logic by using some more powerful logic. Unfortunately, this merely pushes the problem off, since there would remain an obligation to demonstrate that the "more powerful" logic was itself sound. Eventually, one of these logics would have to be justified without resort to a formal proof. Clearly, some level of informal analysis is unavoidable in the pursuit of highly trustworthy systems.

In practice, the more powerful logic is one such as set theory that has been accepted to be sound through (informal) community review. This logic is then used to justify the soundness of the theorem proving system in a non-automated proof[1].

[1] Automation is *not* required for formality, but it does provide for more reliable checking of details.

1.2 Practical Goal

Full (formal, automated or neither) analysis of a theorem proving system, however, would be a decidedly painful experience. Such systems are typically large, and include many features which have little relation to the soundness of the system. Consequently, the practical ideal which has developed in the community is of a system which is constructed in two main layers: 1) a small soundness relevant core, and 2) the rest of the system features wrapped around the core. This reduces the scale of the target of informal analysis to a manageable size.

A second possible benefit to a manage-ably-sized (for non-automated analysis) core is that the hand proof could be published and be vetted by the community at large. Unless the core itself is reasonably small, it is unlikely that the soundness demonstration would be broadly accessible (even to experts in the field).

A third benefit to a system with a small core of primitives is that the system could be designed to output the primitive proof steps, allowing a complete separation between the proof discovery and proof checking aspects of the system. This would provide compelling justification that only the (checking) core of the system would require informal analysis.

Even better, once such a checker had been built and vetted by the community at large, any proposition which had its proof checked by the primitive checker would have been effectively vetted by the community as well. This scheme would maximize the leverage which could be obtained from the investment needed to achieve the second benefit.

1.3 Plan for Realization

Fortunately, HOL is a system designed with this ideal in mind. The logic itself is based on a few axioms, rules of inference and principles of definition. The soundness of this logic is justified by a proof sketch, in chapters 15 and 16 of the HOL documentation[1]. Unfortunately, the system as originally implemented was not as simple or direct as the logic design. Many of the inference rules (for which justifications in terms of the primitive rules are well-documented) were implemented as primitives for efficiency reasons.

This was one of the key motivators for the use of the HOL Light (HOLL) [2] system as the basis of this effort. Only those things which are documented as primitives are actually implemented as primitives. The implementation of the core of the system (except for two rules which are presented later, merely to make the presentation flow smoothly) requires only about 1200 lines of ML code—most of the first 500 or so are just simple library functions [2]. The extension of the original proof sketch (for the revised core) of correctness is documented for this system [3].

[2] wHOLe includes another approximately 400 lines of library code designed to facilitate the translation from CAML to SML

2 EXTENSION

The system presented here is layered on top of the HOLL system, so we will only present justifications for the soundness of the extensions to HOLL. We have extended the core with only two rules:

1. a primitive rule of inference which constructs new theorems by applying one of a set of "certified" pieces of ML code to an existing theorem

```
fun USE_PR n (Sequent(asl,c)) =
    if (can (assoc n) (!valid_rules))
    then (Sequent (asl, (assoc n (!valid_rules)) c))
    else raise Failure "No rule for "^n^" defined.";
```

2. an extension rule which checks that proposed pieces of ML code have the desired property, and if so, adds the code to the "certified" set

```
  fun mk_proof_rule name text prf =
  let val abs_fun = hol_parse text
       val oper = X ("APPexp_e ("^abs_fun^")")
       val rando = X "(PARatexp_e (x:exp_e))"
       val rhs = Comb(oper,rando)
       val x = Var("x",Tyapp("exp_e",[]))
       val (Comb (Const ("SOME_My:A->A option_My"),"result:A")) =
             (mk_comb(X "compute_exp s_i E'_0",rhs))
       val () = mk_comb(X "FST:(val_pack#state)->val_pack",

       val goal = mk_forall(x,(mk_eq(x,result)))
      val func = EVAL.eval text
  in
      (PROVE(goal,prf) handle Failure _ =>
raise Failure "Function not truth preserving!";
      if (can (assoc name) (!valid_rules))
      then raise Failure "Rule Name already in Use"
      else (valid_rules := (name,func)::(!valid_rules);
   (rule_just := (name,prf)::(!rule_just));
   TextIO.output(TextIO.stdOut,
"New proof rule "^name^" has been accepted\n")))
   end;

(``X`` is the function which parses strings into hol_terms)
```

Clearly, these two rules are closely related in their impact on soundness, since the first rule merely applies the code "certified" by the second rule. They are built on top of two simple list structures which store the rules which have been certified and their proofs:

```
val valid_rules = ref []: (string*(term->term)) list ref

val rule_just = ref []: (string*tactic) list ref
```

2.1 Application Rule

Assuming we are able to satisfactorily establish that the extension rule only adds code/proofs to these lists if the code is a sound extension, then the application of the "USE_PR" rule is trivially (theoretically) sound. It does, however, raise two interesting related issues: misuse of the rule structure by the user, and the ability of the resulting system to support an isolated primitive proof checker.

Misuse Two sorts of misuse should be considered: accidental and malicious misuse. We consider the risks of accidental misuse extremely low. The user interface (essentially the traditional HOL interface) to the system is entirely via function applications; users would have no motivation to perform assignments on variables which they have not defined (since, even if they determined the need to use a "ref" variable to manage their work, they would reasonably expect assignment to an undefined variable to fail). Even if a user were to freshly define the variable, static scoping would prevent a conflict with the critical structure.

Malicious misuse, on the other hand, would be easily accomplished. As the system is currently implemented, a brief reading of the code (or this document) would provide the name of the key structure, improper code could be added by simple assignment. Even so, we consider the risk of (successful) malicious use very low:

1. A completed proof is not (currently) an economic commodity; any gain that is achievable in this way is unlikely to be sufficient motivation for the risk of personal embarrassment.
2. Checking that such a deception had been attempted would be quite trivial: any competent reviewer (who examined the proof closely) would note it, or a simple automated scan (e.g. grep) would be more than adequate to detect it.
3. The current implementation is essentially designed to be in "debugging" mode, since its primary purpose is to support this research project—two different techniques could be employed to dramatically decrease the feasibility of malicious misuse, if a production implementation were developed. The first is the straightforward application of the module feature in SML. By placing the basic system inside a module and only allowing the primitive rules to be publicly accessible (thereby hiding the key structures), it would force the malicious user to rewrite the system to achieve such a deception.

Of course, there are a wide variety of ways that a user could get a doctored (rewritten) system to appear to accept bogus proofs—this feature would be just one more target for this attack. Fortunately, the proposal for a split prover (primitive checker/proof assistant) would provide a more than adequate defense for this type of attack (and it's the next issue to be considered).

The Split Prover Finding techniques for creating such a system is still an active research area, recent work includes a proposal for a implementation of such system based on HOL. No effort, consequently, was made to solve a second open

problem or apply such techniques to this system. We claim, though, that these extensions would make the implementation of such techniques no more difficult.

The essential difficulty in the creation of a such a system is reducing an arbitrary proof to one that only makes reference to the primitive rules. Any invocation of one of the code rules by "USE_PR" could be be replaced by instantiating the theorem (∀ t. t = f t) with the conclusion of the current goal, and looking up the proof of this theorem in the second data structure (using the name of the code as an index). Having the precise theorem and its proof should not only be sufficient, but more than is usually available to such a tool. So, while some simple coding may be needed to handle these rules, it is pretty trivial.

2.2 The Extension Rule

This places the obligation to meet the key condition for soundness (i.e. is the justification for these pieces of code sufficient to allow their use) squarely on the second rule. The rule, allowing for three assumptions, has a very straightforward justification. It requires that there is a proof that for any possible HOL term, the result of applying the ML code to that term is equivalent to that term. In any situation where the code rule would be used, a completely ordinary HOL proof is also available. Instead of applying the code rule, the system could instantiate the already proved theorem, and simply rewrite the conclusion of the goal. Since an equivalent primitive proof exists[3], the extension rule is a conservative extension.

The three assumptions that must be justified are: 1) that the parser correctly converts ML code into the appropriate internal form (abstract syntax) 2) the evaluation rules correctly define the operation of SML and 3) the user submits a function object which was created by the system evaluating the string ("text") submitted at the same time.

Parser Fortunately, the construction of a parser was performed with the assistance of automated tools (ML-Lex and ML-Yacc, provided in the SML/NJ distribution). The front end of the parser merely required the straight-forward encoding of the syntax from [5], while the back end ("code generation" of the abstract syntax) is an encoding drawn directly from the holML representation of the formal definition [7]. The tools helped identify a number of minor errors.

Formal Definition—Evaluation Rules The evaluation rules are drawn from HOL-SML, Myra VanInwegen's [7] encoding of the formal definition of SML [5]. HOL-SML includes only the Core of SML, and completely eliminates any of the rules on equality types, real numbers and input/output. A number of the differences between HOL-SML and Core SML '90 (e.g. value restriction versus weak typing,

[3] it may be necessary to reduce subordinate code rule applications to primitives to construct an actual primitive proof

restrictions on the use of the "ref" constructor) needed to make proofs about the language tenable have been incorporated into the SML '97 definition[4].

We have made two small changes to the definition: extending the default SML base environment with the types and constructors for modeling HOL terms (since we want to write SML code which modifies HOL terms); and coercing the variables in two of the pattern phrases into longvars to facilitate parsing. We have only converted the evaluation rule sections of the HOL-SML to the wHOLe system, the elaboration rules, as well as the proof type preservation have not been addressed (work on the determinism proof is underway).

The effort that went into those proofs, however, gives significant credibility to the accuracy of the evaluation rules. Not only was this formal, automated analysis of the SML definition able to achieve interesting results, but had influence on the development of the language. Conversion of the HOL-SML rules for the wHOLe system merely required some syntactic modifications (double quoted strings to represent HOL terms instead of single-quoted, a few more parenthesis to clarify scope of operators)—the difficult part was converting the supporting packages (which exposed numerous distinctions between the HOL '90 and HOLL/wHOLe systems).

The String and the Object While it is trivially easy (using a cut and paste editor) for the user to correctly submit matching text and executable objects, we would prefer to relieve the user of this obligation (and close a pretty large soundness hole). We expect that more careful study of the SML/NJ system will identify a mechanism for handling this detail.

3 USE OF THE SYSTEM

3.1 Basics

Briefly stated, wHOLe is a derivative of the original HOL system. Except for the implementation language, and the modifications described above, it is HOLL. The canonical features of such a system are: that interaction is via a dialect of ML (literally the Meta Language), that backward proof by tactics is the primary metaphor (although other schemes are supported), and the HOL object language (which is Higher Order Logic). Much more detail is available in [1, 4].

It is important to note, however, that HOLL was designed to be a reference version of HOL—the foundation for future research which could feed into system as it developed. HOLL does not include features like libraries or theory management & storage which would be preferred in a production system.

[4] Ironically, there was some trepidation when SML/NJ 93 implementation inadequacies forced this project to upgrade to SML/NJ 110 (and therefore, SML '97)—fearing that it would cause *more* language inconsistencies.

3.2 Extension

A helper function (called prove_SML) has been defined which takes a string (the text of some SML code), and invokes the HOL subgoal package on the theorem ($\forall$ t. t = f t) (where f is the SML code). At this point, the user can interactively apply tactics in the usual way, attempting to reduce the goal to statements that are known to be true. The user is obliged to keep careful track of the tactics employed to prove the goal, so that a single tactic which proves the goal can be constructed (e.g. "tactic1 THEN tactic2" is a single tactic which first applies tactic1, then applies tactic2 to the result).

The user must also compute the executable object equivalent to the text of their function. This can be done by using

> this "val" declaration: val my_sml_function = The Text of the SML Code;
> when the code looks like this: "The Text of the SML Code"

It will create an executable version of the object, and attach to it the name "my_sml_function". We recommend that this be done before the proof of correctness, since we find little value in proving properties of invalid SML code.

We anticipate that proofs of the kind we require will be greatly facilitated by the construction of a "symbolic evaluator" [6] which will generate the tactics to do the obvious case-splitting and rewriting for the analysis of SML code. We have deliberately put off the design of such a tool, however, until we have adequate experience performing such proofs.

4 DISCUSSION

4.1 How it Applies to Practice

Efficiency One of the long-time concerns with theorem proving tools is their speed. Because this tool provides a way to replace proof strategies based on primitive invocations with more efficient direct term manipulation, it should allow for a significant, although probably linear, speed-up.

Paradigm Unification An issue which has been pushed to the forefront by the development of algorithmic methods (e.g. model-checkers), is how to merge analysis tools of varying degrees of formal basis into a single methodology. This tool lays out the path to the ultimate solution to this dilemma: code verification that such an algorithm is truth preserving. While significant improvements to the support for and understanding of the method will be needed before such a proof could be attempted, at least the foundation has been laid.

Life-Cycle and Support The ability of SML/NJ 110 to import and run "C" code (via the CINTERFACE) offers a near-term solution for combining tools and a migration path for achieving the eventual goal. Initially, only the theorem prover would be implemented in SML, while any other tools would just be imported C. To increase the reliability of the tools, they could be a) translated to SML, b) shown to have certain key properties and c) shown to be truth preserving. This plan would allow one to selectively focus resources on the highest risk areas.

4.2 How it Applies to Research

Back-to-Basics Formal verification began as a research field with code techniques developed by Floyd, Hoare and Dijkstra. It is refreshing to be able to provide a modern level of automated support for the techniques on which the field was founded. Even better, the results of these verifications will not be just academic exercises, but can contribute to the development of the tools as well.

Self-Perpetuation The most appealing aspect of a system based on computational reflection via code proofs, though, is that successful application of the system will facilitate even more success. The user can add code rules to the system which will make it faster and more useful. This additional power can be used to incorporate the more sophisticated language features which have not yet been analyzed. The ability to employ more of the language will allow even more code rules to be accepted by the system.

5 ACKNOWLEDGMENTS

Great thanks to Matt Kaufmann, Myra VanInwegen, Richard Boulton, Elsa Gunter, John Reppy, Ian Soboroff, Konrad Slind, Donald Syme, John Harrison.

References

1. Michael J. C. Gordon and Tom F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, Cambridge, UK, 1993.
2. John Harrison. HOL light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.
3. John Harrison. A Tour of HOL light. Unpublished Draft, October 1996.
4. Laboratory for Applied Logic, Brigham Young University, http://lal.cs.byu.edu/lal/hol-documentation.html. *The HOL Theorem Proving System.* Web site featuring manuals, links to tutorials, on-line searchable libraries.
5. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, Cambridge MA USA, 1990.
6. Donald Syme. Reasoning with the formal definition of Standard ML in HOL. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications (HUG'93); 6th International Workshop*, volume 780 of *Lecture Notes in Computer Science*, pages 43–58. Springer-Verlag, 1993. Vancouver, BC, CANADA.
7. Myra VanInwegen. *The Machine-Assisted Proof of Programming Language Properties.* PhD thesis, University of Pennsylvania, August 1996.

Z/EVES Version 1.5: An Overview

ORA Canada

1208 One Nicholas Street
Ottawa, Ontario, K1N 7B7
Canada

1 System Overview

Z/EVES supports the analysis of Z specifications in several different ways:

- syntax and type checking,
- schema expansion,
- precondition calculation,
- domain checking (are expressions meaningful?),
- symbolic execution,
- refinement proofs, and
- general theorem proving.

The range of analysis supports an incremental introduction of Z/EVES capabilities to new users. For example, very little knowledge of the theorem prover is required for syntax and type checking, schema expansion, symbolic execution, and precondition calculation. Even with domain checking, many of the proof obligations are easily proven; and for those that are not, often the generation of the proof obligation is a substantial aid in determining whether a meaningful specification has been written. So far, every substantial Z specification that we have analyzed, has been shown to have some incorrect function applications.

Z/EVES integrates a leading specification notation with a leading automated deduction capability. Z/EVES supports almost the entire Z notation; only the unique existence quantifier in schema expressions is not currently supported. The Z/EVES prover provides powerful automated support (*e.g.*, conditional rewriting, heuristics, decision procedures) with user commands that have been found to be helpful in directing the prover in specific ways (*e.g.*, instantiate a specific variable, introduce a lemma, use a function definition). An extended version of the Z Mathematical Toolkit has been developed and is included with the release. Extensive effort has been directed at automating as much of the Toolkit as possible.

Interaction with the Z/EVES system uses an extension of the "fuzz" syntax, which is based on LaTeX. Our extensions add a means for attaching labels to predicates in an axiomatic or generic box; a paragraph that expresses a theorem; and prover commands. Z/EVES can read files that have been prepared for the fuzz typechecker.

2 Proving theorems with Z/EVES

Except for syntax and type checking, all the analyses mentioned above are based on the Z/EVES theorem proving capability. We briefly describe this capability before illustrating the analyses possible in Z/EVES.

2.1 Starting proofs

Proofs in Z/EVES can be started in four different ways: the **try** command can be used to state a predicate to try proving; a **theorem** paragraph states a theorem and begins its proof; the **try lemma** command can be used to continue the proof of some previously defined theorem; and entering a paragraph can begin a proof of an associated *domain condition*, which is a formula that guarantees that the expressions in that paragraph are well-defined (avoiding, for example, domain errors such as division by zero).

A goal is just a Z predicate. Theorems and goals are allowed to refer to free variables—that is, variables that are not bound by any quantifiers.

2.2 The structure of proofs

A proof in Z/EVES is a sequence of steps, each of which transforms the current goal predicate into a logically equivalent predicate. When the goal has been transformed to the predicate *true*, the proof is complete.

Z/EVES provides a spectrum of commands, ranging from small manipulations of the formula to powerful heuristically-driven proof procedures. These procedures can use theorems that the user has marked as rules (for rewriting), frules (for forward chaining inferences), or grules (for triggered assumptions). The details of the proof commands and proof mechanisms are described in the Z/EVES reference manual.

For the examples here, we use the most powerful of the automatic commands: **prove by reduce**, which directs the prover to expand definitions of schemas and abbreviations and to use rewrite rules, equality substitution, and propositional simplification.

3 Domain checking

For each paragraph in a specification, Z/EVES derives a *domain condition*. This domain condition is a predicate asserting that all the expressions appearing in the paragraph are meaningful. There are two ways an expression can fail to have a meaning: first, by applying a partial function to an element not in its domain, and second, by using an improper definite description.

If a paragraph has a nontrivial domain condition, Z/EVES generates a domain check conjecture for it. If this conjecture can be proved, then the paragraph has a well-defined meaning.

Consider the following specification, which describes a simple personnel database that records a set of employees, and for each employee records their boss and their salary. There are two constraints: each employee has a salary, and an employee's salary is less than their boss' salary.

$[PERSON]$

Personnel

$$
\begin{array}{l}
employees : \mathbb{P}\, PERSON \\
boss_of : PERSON \pfun PERSON \\
salary : PERSON \pfun \mathbb{N} \\
\hline
\mathrm{dom}\, salary = employees \\
\forall e : employees \bullet salary(e) < salary(boss_of\ e)
\end{array}
$$

A non-trivial domain checking condition is generated for Schema *Personnel*:

$$
\begin{array}{l}
\quad employees \in \mathbb{P}\, PERSON \\
\quad \land\ boss_of \in PERSON \pfun PERSON \\
\quad \land\ salary \in PERSON \pfun \mathbb{N} \\
\quad \land\ \mathrm{dom}\, salary = employees \\
\quad \land\ e \in employees \\
\Rightarrow e \in \mathrm{dom}\, salary \\
\quad \land\ e \in \mathrm{dom}\, boss_of \\
\quad \land\ boss_of\ e \in \mathrm{dom}\, salary
\end{array}
$$

The three conjuncts of the conclusion of this formula correspond to the three function applications in the last predicate of the schema.

We can simplify the condition somewhat by using a **rewrite** or **reduce** command; here we will rewrite, which produces the formula

$$
\begin{array}{l}
\quad employees \in \mathbb{P}\, PERSON \\
\quad \land\ boss_of \in PERSON \pfun PERSON \\
\quad \land\ salary \in PERSON \pfun \mathbb{N} \\
\quad \land\ \mathrm{dom}\, salary = employees \\
\quad \land\ e \in employees \\
\Rightarrow e \in \mathrm{dom}\, boss_of \\
\quad \land\ boss_of\ e \in \mathrm{dom}\, salary
\end{array}
$$

Z/EVES has not made much progress, and for a good reason: this goal is not a theorem. The condition shows that we have forgotten some conditions in the schema. Both $e \in \mathrm{dom}\, boss_of$ and $boss_of\ e \in \mathrm{dom}\, salary$ concern the well-definedness of the final condition in the schema, $\forall e : employees \bullet salary(e) < salary(boss_of\ e)$. Firstly, not every employee has a boss (in particular, the president would not). Our specification states only that *boss_of* is some partial function, without specifying its domain. So, in the quantification, e should be

specified to range only over those employees with bosses. Secondly, *boss_of e* might not be in the domain of *salary* (which we know is the set of employees). Indeed, we have forgotten to specify that the range of *boss_of* includes only employees.

As this example shows, domain conditions often fail when a specifier has been sloppy. The domain checking analysis therefore provides a useful error screen that goes beyond type checking to uncover semantic errors in a specification.

The Z/EVES approach to domain errors is conservative; different Z users adopt different conventions. One common convention is to treat any atomic formula containing an expression with a domain error is simply false. This works fine for the library example above, but can lead to surprises in other specifications. The draft Z Standard leaves the meaning of expressions containing domain errors unspecified; our conservative approach therefore seems to be the safest treatment.

4 Schema expansion

The schema calculus, and schema inclusion, allow for very compact descriptions of systems. Sometimes, though, these descriptions are rather too compact. Z/EVES can be used to expand schemas and simplify the result, showing the full meaning of a schema. For example, Consider the following specification of a simple banking system:

Account

$$overdraft_limit : \mathbb{N}$$
$$balance : \mathbb{Z}$$
$$balance + overdraft_limit \geq 0$$

OpenAccount

$$Account'$$
$$overdraft_limit? : \mathbb{N}$$
$$overdraft_limit' = overdraft_limit?$$
$$balance' = 0$$

(other operations on accounts omitted ...)

$[AccountID, CLIENT]$

Bank

$$
\begin{array}{l}
clients : \mathbb{P}\, CLIENT \\
account : AccountID \pfun Account \\
accounts : \mathbb{P}\, AccountID \\
access : CLIENT \rel AccountID \\
\hline
accounts = \mathrm{dom}\, account \\
\mathrm{dom}\, access \subseteq clients \\
\mathrm{ran}\, access \subseteq accounts
\end{array}
$$

NewAccount

$$
\begin{array}{l}
\Delta Bank \\
client? : CLIENT \\
overdraft_limit? : \mathbb{N} \\
account_id! : AccountID \\
\hline
client? \in clients \\
account_id! \notin accounts \\
\exists\, Account' \mid OpenAccount \bullet account' = account \oplus \{account_id! \mapsto \theta Account'\} \\
access' = access \cup \{(client?, account_id!)\}
\end{array}
$$

We can use Z/EVES to expand schema *NewAccount* by starting with it as a goal:

```
try NewAccount;
```

(thus, strictly speaking, we are simplifying the predicate associated with the schema). By using the **prove by reduce** command, we can arrive at the formula

$$
\begin{array}{l}
clients \in \mathbb{P}\, CLIENT \\
\land\ account \in AccountID \pfun Account \\
\land\ accounts = \mathrm{dom}\, account \\
\land\ access \in CLIENT \rel AccountID \\
\land\ clients' \in \mathbb{P}\, CLIENT \\
\land\ overdraft_limit? \in \mathbb{Z} \\
\land\ account_id! \in AccountID \\
\land\ account' = \\
account \oplus \{(account_id!, \theta Account[balance := 0, overdraft_limit?/overdraft_limit])\} \\
\land\ accounts' = \{account_id!\} \cup \mathrm{dom}\, account \\
\land\ client? \in CLIENT \\
\land\ client? \in clients \\
\land\ client? \in clients' \\
\land\ access' = access \cup \{(client?, account_id!)\} \\
\land\ overdraft_limit? \geq 0 \\
\land\ \mathrm{dom}\, access \in \mathbb{P}\, clients \\
\land\ \mathrm{ran}\, access \in \mathbb{P}\, \mathrm{dom}\, account \\
\land\ \mathrm{dom}\, access \in \mathbb{P}\, clients' \\
\land\ \neg\ account_id! \in \mathrm{dom}\, account
\end{array}
$$

This predicate can be examined to ensure that all the expected constraints are present.

5 Consistency Proofs

A variety of consistency proofs are possible for Z specifications: proofs that an axiomatic definition is satisfiable, proof that a free type definition is satisfiable, or proofs that a schema is satisfiable. We will illustrate just schema satisfiability here, although the other proofs are possible.

It is easy to express the satisfiability of a schema S, using the predicate $\exists S \bullet true$. Sometimes a more impressive alternative is available: many specifications give an initialization schema of the form $Init_S \mathrel{\widehat=} [S \mid P]$, where the predicate P further constrains the state. In such a case, showing $\exists S \bullet Init_S$ not only shows that S is satisfiable, it also shows that initial states are possible.

For the banking example presented above, we might add the initialization schema

InitBank

$Bank'$

$clients' = \emptyset$
$account' = \emptyset$
$accounts' = \emptyset$
$access' = \emptyset$

The satisfiability of this initialization schema (and schema *Bank* itself) can be stated as $\exists Bank' \bullet InitBank$ and proven automatically with the command `prove by reduce`. (In cases where some schema components are not given explicit initial values in the initialization schema, it may be necessary to instantiate them manually in the consistency proof.)

6 Symbolic Execution

Z specifications are not necessarily executable, as the full power of first order logic and set theory is available. There is some benefit to be gained, however, by animating a specification. For example, clients can see whether a system's behaviour meets their expectations, and specifiers can detect errors and omissions in a specification by testing it. Some authors have promoted restricted subsets of the notation that can be executed. Z/EVES offers a second choice: the prover can be applied to an instance of an operation, and through various simplifications, can describe the result.

Sequences of operations can be specified using schema composition, so short sequences of operations can be specified and the results determined. For example, continuing the banking specification with the operation of adding a client:

AddClient

$\Delta Bank$
$client? : CLIENT$

$clients' = clients \cup \{client?\}$
$account' = account$
$access' = access$

we can test the sequence of starting a bank, then adding a client, by defining the schema

$$TestBank1 \mathrel{\widehat=} InitBank \fatsemi AddClient$$

an expanding it.

```
try TestBank1;
```

The result of the **prove by reduce** command in the predicate

$$\begin{aligned}
&access' = \{\} \\
&\land\ account' = \{\} \\
&\land\ accounts' = \{\} \\
&\land\ client? \in CLIENT \\
&\land\ clients' = \{client?\}
\end{aligned}$$

Defining

$$TestBank2 \mathrel{\widehat=} TestBank1 \fatsemi NewAccount$$

lets us extend the scenario; trying *TestBank2* and issuing the verb'prove by reduce' command results in

$$\begin{aligned}
&account_id! \in AccountID \\
&\land\ client? \in CLIENT \\
&\land\ clients' \in \mathbb{P}\, CLIENT \\
&\land\ client? \in clients' \\
&\land\ access' = \{(client?, account_id!)\} \\
&\land\ overdraft_limit? \in \mathbb{Z} \\
&\land\ account' \\
&= \{(account_id!, \theta Account[balance := 0, overdraft_limit?/overdraft_limit])\} \\
&\land\ accounts' = \{account_id!\} \\
&\land\ overdraft_limit? \geq 0
\end{aligned}$$

Scrutiny of this predicate shows an anomaly: the value of $clients'$ is relatively unconstrained (the relavent conjuncts are $clients' \in \mathbb{P}\, CLIENT$ and $client? \in clients'$), while we expected $clients' = \{client?\}$. This suggests a problem in the specification, and indeed it is readily seen that the constraint $clients' = clients$ was omitted from schema *NewAccount*.

7 Precondition calculation

Z/EVES can be used to calculate the precondition of a schema. Given a state schema S and an operation declared by the schema

$$
\begin{array}{|l}
\hline
OP \\
S \\
S' \\
in? : X \\
out! : Y \\
\hline
condition \\
\hline
\end{array}
$$

the precondition is defined to be the formula $\exists S'; out! : Y \bullet Op$, which guarantees that there is a possible output and final state. It is most convenient to work with the formula $S \land in? \in X \Rightarrow \exists S'; out! : Y \bullet Op$, which includes the reasonable assumption that the initial state and input are suitable. This formula gives the "missing" part of the precondition (and, if the operation is total, is equivalent to *true*). Z/EVES can be used to simplify this predicate.

For example, a City Hall marriage registry might contain a record of all married couples:

$[Man, Woman]$

The relation *wie_of* records this information, it is a partial (not all men are married) injection (bigamy is illegal) from men to women.

$$
\begin{array}{|l}
\hline
Registry \\
wife_of : Man \rightarrowtail Woman \\
\hline
\end{array}
$$

The *Wed* operation adds new couple to the registry:

$$
\begin{array}{|l}
\hline
Wed \\
\Delta Registry \\
bride? : Woman \\
groom? : Man \\
\hline
wife_of' = wife_of \cup \{groom? \mapsto bride?\} \\
\hline
\end{array}
$$

Obviously, this is not a total operation. We can explore the precondition by working on the formula

$$Registry \land bride? \in Woman \land groom? \in Man \Rightarrow \text{pre } Wed.$$

The first **prove by reduce** command gives

$$
\begin{array}{l}
wife_of \in Man \rightarrowtail Woman \\
\land\ bride? \in Woman \\
\land\ groom? \in Man \\
\Rightarrow wife_of \cup \{(groom?, bride?)\} \in Man \rightarrowtail Woman
\end{array}
$$

We can manually apply a Toolkit theorem about unions being partial injections; then **prove by reduce** gives the predicate

$$wife_of \in Man \rightarrowtail Woman \land bride? \in Woman \land groom? \in Man$$
$$\Rightarrow (groom? \in \mathrm{dom}\, wife_of \Rightarrow wife_of(groom?) = bride?)$$
$$\land\ (bride? \in \mathrm{ran}\, wife_of \Rightarrow wife_of^{\sim}(bride?) = groom?)$$

As can be seen, if either of the bride or groom was already married, the two must already be a couple.

8 Theorem proving

Specifications can be validated by *challenge theorems*. These are facts that should be consequences of the specification, but are not stated explicitly. Z/EVES allows for the statement of such theorems, and can be used to try to prove them.

Here is a simple example for the registry. We have decided to make the *Wed* operation applicable only for couples who are not already married:

Wed

$\Delta Registry$
$bride? : Woman$
$groom? : Man$

$bride? \notin \mathrm{ran}\, wife_of$
$groom? \notin \mathrm{dom}\, wife_of$
$wife_of' = wife_of \cup \{groom? \mapsto bride?\}$

Obviously, after a wedding, the bride should be recorded as the wife of the groom. We can check that by proving $\forall\ Wed \bullet wife_of'(groom?) = bride?$. A **prove by reduce** command completes the proof automatically.

9 Applications

Z/EVES was first released in 1996. Since then it has been distributed to over 300 sites and has been used on a number of reasonably large specifications for type checking, schema expansion and domain analysis. We are aware of plans for both the commercial and academic use of Z/EVES. Z/EVES will be used to teach Z and/or theorem proving at a number of universities. The authors of several technical reports and theses have applied Z/EVES to their Z specifications.

10 Z Browser

A Dynamic Data Exchange (DDE) link between Z/EVES and the Slovakian Z Browser tool has been implemented. Separate acquisition of Z Browser will allow for browsing Z declarations in a Z font as they are added to Z/EVES. Information on Z Browser is obtainable from the ORA Canada web site. Z Browser is a PC-based tool. (Z Browser is not essential to the productive use of Z/EVES.)

11 Platforms

Z/EVES runs on SunOS, OS/2, Linux, Windows 3.1/95/98/NT, and, with the appropriate computability package from Sun, Solaris.

12 Release Procedure

Z/EVES is currently being released electronically. Though Z/EVES is being released at no cost (for an electronic distribution), prospective users of Z/EVES must sign a user license. Full details of the release procedure are described at our web site (see below).

13 More information

For further information on Z/EVES, including technical reports and release procedure, visit our Z/EVES web page,

http://www.ora.on.ca/z-eves/welcome.html

A Z/EVES interest mailing list has been created. To subscribe, send electronic mail with the body consisting only of the word **subscribe** to **zeves-request@ora.on.ca**.

Electronic mail can also be send to **ora@ora.on.ca**.

Author Index

Agerholm S. 168,326
Almeida M.J.B. 77

Balser M. 330,351
Börger E. 1
Bogdanov K. 107
Broy M. 44
Büssow R. 184

de Camargo M.S. 77
Clement T. 296

Del Castillo G. 311
Domingos M. 77
Dufourd J.-F. 62

Farines J.-M. 77
Fantechi A. 228
Fujita M. 281

Gaspary L.P. 77
Geser A. 92
Gnesi S. 228
Goerigk W. 122
Goldsmith M. 243
Granville L.Z. 77
Grieskamp W. 184
Gruhn V. 213

Heicking W. 184
Herrmann S. 184
Hoffmann U. 122
Holcombe M. 107
Hu A. 281
Hutter D. 351

Karlsen E.W. 266
Koob F. 302
Krieg-Brückner B. 251
Küchlin, W. 92
Kutter P.W. 196

Larsen P.G. 168,326
Lopes de Souza, W. 77

Mantel H. 351
Margaria T. 213
Mazzanti F. 228
van der Meulen M. 296

Owre S. 338
ORA 367

Pnueli A. 137
Pugliese R. 228
Puitg F. 62

Rajan S.P. 281
Reif W. 330,351
Rock G. 351
Rushby J.M. 338

Sampaio P.N.M. 77
Scheffel R.M. 77
Schellhorn G. 330,351
Schweizer D. 196
Shankar N. 338
Shtrichman O. 137
Siegel M. 137
Singh H. 107
Slotosch O. 44,346
Stenzel K. 330,351
Stephan W. 351
Stringer-Calvert, D.W.J. 338

Thiele L. 196
Tronci E. 228

Ullmann M. 302

Willrich R. 77
Wittmann S. 302
Wolpers A. 351
Woodcock M.E. 359

Yamane S. 151

Zakiuddin I. 243

Lecture Notes in Computer Science

For information about Vols. 1–1594
please contact your bookseller or Springer-Verlag

Vol. 1595: K. Hammond, T. Davie, C. Clack (Eds.), Implementation of Functional Languages. Proceedings, 1998. X, 247 pages. 1999.

Vol. 1596: R. Poli, H.-M. Voigt, S. Cagnoni, D. Corne, G.D. Smith, T.C. Fogarty (Eds.), Evolutionary Image Analysis, Signal Processing and Telecommunications. Proceedings, 1999. X, 225 pages. 1999.

Vol. 1597: H. Zuidweg, M. Campolargo, J. Delgado, A. Mullery (Eds.), Intelligence in Services and Networks. Proceedings, 1999. XII, 552 pages. 1999.

Vol. 1598: R. Poli, P. Nordin, W.B. Langdon, T.C. Fogarty (Eds.), Genetic Programming. Proceedings, 1999. X, 283 pages. 1999.

Vol. 1599: T. Ishida (Ed.), Multiagent Platforms. Proceedings, 1998. VIII, 187 pages. 1999. (Subseries LNAI).

Vol. 1600: M. Wooldridge, M. Veloso (Eds.), Artificial Intelligence Today. VIII, 489 pages. 1999. (Subseries LNAI).

Vol. 1601: J.-P. Katoen (Ed.), Formal Methods for Real-Time and Probabilistic Systems. Proceedings, 1999. X, 355 pages. 1999.

Vol. 1602: A. Sivasubramaniam, M. Lauria (Eds.), Network-Based Parallel Computing. Proceedings, 1999. VIII, 225 pages. 1999.

Vol. 1603: J. Vitek, C.D. Jensen (Eds.), Secure Internet Programming. X, 501 pages. 1999.

Vol. 1604: M. Asada, H. Kitano (Eds.), RoboCup-98: Robot Soccer World Cup II. XI, 509 pages. 1999. (Subseries LNAI).

Vol. 1605: J. Billington, M. Diaz, G. Rozenberg (Eds.), Application of Petri Nets to Communication Networks. IX, 303 pages. 1999.

Vol. 1606: J. Mira, J.V. Sánchez-Andrés (Eds.), Foundations and Tools for Neural Modeling. Proceedings, Vol. I, 1999. XXIII, 865 pages. 1999.

Vol. 1607: J. Mira, J.V. Sánchez-Andrés (Eds.), Engineering Applications of Bio-Inspired Artificial Neural Networks. Proceedings, Vol. II, 1999. XXIII, 907 pages. 1999.

Vol. 1608: S. Doaitse Swierstra, P.R. Henriques, J.N. Oliveira (Eds.), Advanced Functional Programming. Proceedings, 1998. XII, 289 pages. 1999.

Vol. 1609: Z. W. Raś, A. Skowron (Eds.), Foundations of Intelligent Systems. Proceedings, 1999. XII, 676 pages. 1999. (Subseries LNAI).

Vol. 1610: G. Cornuéjols, R.E. Burkard, G.J. Woeginger (Eds.), Integer Programming and Combinatorial Optimization. Proceedings, 1999. IX, 453 pages. 1999.

Vol. 1611: I. Imam, Y. Kodratoff, A. El-Dessouki, M. Ali (Eds.), Multiple Approaches to Intelligent Systems. Proceedings, 1999. XIX, 899 pages. 1999. (Subseries LNAI).

Vol. 1612: R. Bergmann, S. Breen, M. Göker, M. Manago, S. Wess, Developing Industrial Case-Based Reasoning Applications. XX, 188 pages. 1999. (Subseries LNAI).

Vol. 1613: A. Kuba, M. Šámal, A. Todd-Pokropek (Eds.), Information Processing in Medical Imaging. Proceedings, 1999. XVII, 508 pages. 1999.

Vol. 1614: D.P. Huijsmans, A.W.M. Smeulders (Eds.), Visual Information and Information Systems. Proceedings, 1999. XVII, 827 pages. 1999.

Vol. 1615: C. Polychronopoulos, K. Joe, A. Fukuda, S. Tomita (Eds.), High Performance Computing. Proceedings, 1999. XIV, 408 pages. 1999.

Vol. 1616: P. Cointe (Ed.), Meta-Level Architectures and Reflection. Proceedings, 1999. XI, 273 pages. 1999.

Vol. 1617: N.V. Murray (Ed.), Automated Reasoning with Analytic Tableaux and Related Methods. Proceedings, 1999. X, 325 pages. 1999. (Subseries LNAI).

Vol. 1618: J. Bézivin, P.-A. Muller (Eds.), The Unified Modeling Language. Proceedings, 1998. IX, 443 pages. 1999.

Vol. 1619: M.T. Goodrich, C.C. McGeoch (Eds.), Algorithm Engineering and Experimentation. Proceedings, 1999. VIII, 349 pages. 1999.

Vol. 1620: W. Horn, Y. Shahar, G. Lindberg, S. Andreassen, J. Wyatt (Eds.), Artificial Intelligence in Medicine. Proceedings, 1999. XIII, 454 pages. 1999. (Subseries LNAI).

Vol. 1621: D. Fensel, R. Studer (Eds.), Knowledge Acquisition Modeling and Management. Proceedings, 1999. XI, 404 pages. 1999. (Subseries LNAI).

Vol. 1622: M. González Harbour, J.A. de la Puente (Eds.), Reliable Software Technologies – Ada-Europe'99. Proceedings, 1999. XIII, 451 pages. 1999.

Vol. 1623: T. Reinartz, Focusing Solutions for Data Mining. XV, 309 pages. 1999. (Subseries LNAI).

Vol. 1625: B. Reusch (Ed.), Computational Intelligence. Proceedings, 1999. XIV, 710 pages. 1999.

Vol. 1626: M. Jarke, A. Oberweis (Eds.), Advanced Information Systems Engineering. Proceedings, 1999. XIV, 478 pages. 1999.

Vol. 1627: T. Asano, H. Imai, D.T. Lee, S.-i. Nakano, T. Tokuyama (Eds.), Computing and Combinatorics. Proceedings, 1999. XIV, 494 pages. 1999.

Col. 1628: R. Guerraoui (Ed.), ECOOP'99 - Object-Oriented Programming. Proceedings, 1999. XIII, 529 pages. 1999.

Vol. 1629: H. Leopold, N. García (Eds.), Multimedia Applications, Services and Techniques - ECMAST'99. Proceedings, 1999. XV, 574 pages. 1999.

Vol. 1631: P. Narendran, M. Rusinowitch (Eds.), Rewriting Techniques and Applications. Proceedings, 1999. XI, 397 pages. 1999.

Vol. 1632: H. Ganzinger (Ed.), Automated Deduction – Cade-16. Proceedings, 1999. XIV, 429 pages. 1999. (Subseries LNAI).

Vol. 1633: N. Halbwachs, D. Peled (Eds.), Computer Aided Verification. Proceedings, 1999. XII, 506 pages. 1999.

Vol. 1634: S. Džeroski, P. Flach (Eds.), Inductive Logic Programming. Proceedings, 1999. VIII, 303 pages. 1999. (Subseries LNAI).

Vol. 1636: L. Knudsen (Ed.), Fast Software Encryption. Proceedings, 1999. VIII, 317 pages. 1999.

Vol. 1637: J.P. Walser, Integer Optimization by Local Search. XIX, 137 pages. 1999. (Subseries LNAI).

Vol. 1638: A. Hunter, S. Parsons (Eds.), Symbolic and Quantitative Approaches to Reasoning and Uncertainty. Proceedings, 1999. IX, 397 pages. 1999. (Subseries LNAI).

Vol. 1639: S. Donatelli, J. Kleijn (Eds.), Application and Theory of Petri Nets 1999. Proceedings, 1999. VIII, 425 pages. 1999.

Vol. 1640: W. Tepfenhart, W. Cyre (Eds.), Conceptual Structures: Standards and Practices. Proceedings, 1999. XII, 515 pages. 1999. (Subseries LNAI).

Vol. 1641: D. Hutter, W. Stephan, P. Traverso, M. Ullmann (Eds.), Applied Formal Methods – FM-Trends 98. Proceedings, 1998. XI, 377 pages. 1999.

Vol. 1642: D.J. Hand, J.N. Kok, M.R. Berthold (Eds.), Advances in Intelligent Data Analysis. Proceedings, 1999. XII, 538 pages. 1999.

Vol. 1643: J. Nešetřil (Ed.), Algorithms – ESA '99. Proceedings, 1999. XII, 552 pages. 1999.

Vol. 1644: J. Wiedermann, P. van Emde Boas, M. Nielsen (Eds.), Automata, Languages, and Programming. Proceedings, 1999. XIV, 720 pages. 1999.

Vol. 1645: M. Crochemore, M. Paterson (Eds.), Combinatorial Pattern Matching. Proceedings, 1999. VIII, 295 pages. 1999.

Vol. 1647: F.J. Garijo, M. Boman (Eds.), Multi-Agent System Engineering. Proceedings, 1999. X, 233 pages. 1999. (Subseries LNAI).

Vol. 1648: M. Franklin (Ed.), Financial Cryptography. Proceedings, 1999. VIII, 269 pages. 1999.

Vol. 1649: R.Y. Pinter, S. Tsur (Eds.), Next Generation Information Technologies and Systems. Proceedings, 1999. IX, 327 pages. 1999.

Vol. 1650: K.-D. Althoff, R. Bergmann, L.K. Branting (Eds.), Case-Based Reasoning Research and Development. Proceedings, 1999. XII, 598 pages. 1999. (Subseries LNAI).

Vol. 1651: R.H. Güting, D. Papadias, F. Lochovsky (Eds.), Advances in Spatial Databases. Proceedings, 1999. XI, 371 pages. 1999.

Vol. 1652: M. Klusch, O.M. Shehory, G. Weiss (Eds.), Cooperative Information Agents III. Proceedings, 1999. XI, 404 pages. 1999. (Subseries LNAI).

Vol. 1653: S. Covaci (Ed.), Active Networks. Proceedings, 1999. XIII, 346 pages. 1999.

Vol. 1654: E.R. Hancock, M. Pelillo (Eds.), Energy Minimization Methods in Computer Vision and Pattern Recognition. Proceedings, 1999. IX, 331 pages. 1999.

Vol. 1656: S. Chatterjee, J.F. Prins, L. Carter, J. Ferrante, Z. Li, D. Sehr, P.-C. Yew (Eds.), Languages and Compilers for Parallel Computing. Proceedings, 1998. XI, 384 pages. 1999.

Vol. 1661: C. Freksa, D.M. Mark (Eds.), Spatial Information Theory. Proceedings, 1999. XIII, 477 pages. 1999.

Vol. 1662: V. Malyshkin (Ed.), Parallel Computing Technologies. Proceedings, 1999. XIX, 510 pages. 1999.

Vol. 1663: F. Dehne, A. Gupta. J.-R. Sack, R. Tamassia (Eds.), Algorithms and Data Structures. Proceedings, 1999. IX, 366 pages. 1999.

Vol. 1664: J.C.M. Baeten, S. Mauw (Eds.), CONCUR'99. Concurrency Theory. Proceedings, 1999. XI, 573 pages. 1999.

Vol. 1666: M. Wiener (Ed.), Advances in Cryptology – CRYPTO '99. Proceedings, 1999. XII, 639 pages. 1999.

Vol. 1668: J.S. Vitter, C.D. Zaroliagis (Eds.), Algorithm Engineering. Proceedings, 1999. VIII, 361 pages. 1999.

Vol. 1671: D. Hochbaum, K. Jansen, J.D.P. Rolim, A. Sinclair (Eds.), Randomization, Approximation, and Combinatorial Optimization. Proceedings, 1999. IX, 289 pages. 1999.

Vol. 1672: M. Kutylowski, L. Pacholski, T. Wierzbicki (Eds.), Mathematical Foundations of Computer Science 1999. Proceedings, 1999. XII, 455 pages. 1999.

Vol. 1673: P. Lysaght, J. Irvine, R. Hartenstein (Eds.), Field Programmable Logic and Applications. Proceedings, 1999. XI, 541 pages. 1999.

Vol. 1674: D. Floreano, J.-D. Nicoud, F. Mondada a(Eds.), Advances in Artificial Life. Proceedings, 1999. XVI, 737 pages. 1999. (Subseries LNAI).

Vol. 1677: T. Bench-Capon, G. Soda, A M. Tjoa (Eds.), Database and Expert Systems Applications. Proceedings, 1999. XVIII, 1105 pages. 1999.

Vol. 1678: M.H. Böhlen, C.S. Jensen, M.O. Scholl (Eds.), Spatio-Temporal Database Management. Proceedings, 1999. X, 243 pages. 1999.

Vol. 1684: G. Ciobanu, G. Păun (Eds.), Fundamentals of Computation Theory. Proceedings, 1999. XI, 570 pages. 1999.

Vol. 1685: P. Amestoy, P. Berger, M. Daydé, I. Duff, V. Frayssé, L. Giraud, D. Ruiz (Eds.), Euro-Par'99. Parallel Processing. Proceedings, 1999. XXXII, 1503 pages. 1999.

Vol. 1688: P. Bouquet, L. Serafini, P. Brézillon, M. Benerecetti, F. Castellani (Eds.), Modeling and Using Context. Proceedings, 1999. XII, 528 pages. 1999. (Subseries LNAI).

Vol. 1689: F. Solina, A. Leonardis (Eds.), Computer Analysis of Images and Patterns. Proceedings, 1999. XIV, 650 pages. 1999.

Vol. 1690: Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, L. Théry (Eds.), Theorem Proving in Higher Order Logics. Proceedings, 1999. VIII, 359 pages. 1999.

Vol. 1694: A. Cortesi, G. Filé (Eds.), Static Analysis. Proceedings, 1999. VIII, 357 pages. 1999.