Tiziana Margaria
Bernhard Steffen (Eds.)

LNCS 4313

Leveraging Applications of Formal Methods

First International Symposium, ISoLA 2004
Paphos, Cyprus, October/November 2004
Revised Selected Papers

Springer

Lecture Notes in Computer Science 4313

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Tiziana Margaria Bernhard Steffen (Eds.)

Leveraging Applications of Formal Methods

First International Symposium, ISoLA 2004
Paphos, Cyprus, October 30 - November 2, 2004
Revised Selected Papers

Springer

Volume Editors

Tiziana Margaria
Lehrstuhl für Service and Software Engineering
Universität Potsdam , Germany
E-mail: margaria@cs.uni-potsdam.de

Bernhard Steffen
Universität Dortmund
FB Informatik, Lehrstuhl 5
Dortmund, Germany
E-mail: steffen@cs.uni-dortmund.de

Library of Congress Control Number: 2006935874

CR Subject Classification (1998): F.3, D.2.4, D.3, C.3, D.2

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-540-48928-2 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-48928-3 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springer.com

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11925040 06/3142 5 4 3 2 1 0

Preface

This volume contains the main proceedings for ISoLA 2004, the 1st International Symposium on Leveraging Applications of Formal Methods held in Paphos (Cyprus) October–November 2004. Besides the 12 papers in this volume, other ISoLA 2004 contributions were selected for thematical special issues of the international journals of *Theoretical Computer Science* (TCS-B), *Software Tools for Technolgoy Transfer* (STTT), as well as *Integrated Design and Process Science* (SDPS transactions).

ISoLA 2004 served the need of providing a forum for developers, users, and researchers to discuss issues related to the adoption and use of rigorous tools and methods for the specification, analysis, verification, certification, construction, test, and maintenance of systems from the point of view of their different application domains. Thus, the ISoLA series of events serves the purpose of bridging the gap between designers and developers of rigorous tools, and users in engineering and in other disciplines, and of fostering and exploiting synergetic relationships among scientists, engineers, software developers, decision makers, and other critical thinkers in companies and organizations. In particular, by providing a venue for the discussion of common problems, requirements, algorithms, methodologies, and practices, ISoLA aims at supporting researchers in their quest to improve the utility, reliability, flexibility, and efficiency of tools for building systems, and users in their search for adequate solutions to their problems.

September 2006

Tiziana Margaria
Bernhard Steffen

Organization

Committees

Symposium Chair	Tiziana Margaria, University of Potsdam, Germany
Program Chair	Bernhard Steffen, University of Dortmund, Germany
Organization Chair	Anna Philippou, University of Cyprus, Cyprus
Industrial Chair	Manfred Reitenspiess, Fujitsu Siemens, Germany

Program Committee

Ed Brinksma	University of Twente, Netherlands
Peter Buchholz	University of Dortmund, Germany
Muffy Calder	University of Glasgow, UK
Radhia Cousot	Ecole Polytechnique, France
Jin Song Dong	National University of Singapore
Ibrahim Esat	Brunel University, UK
John Fitzgerald	University of Newcastle, UK
Robert Giegerich	University of Bielefeld, Germany
Joshua Guttman	MITRE, USA
John Hatcliff	Kansas State University, USA
Mats Heimdahl	University of Minnesota, USA
Joost-Pieter Katoen	University of Twente, Netherlands
Jens Knoop	TU Vienna, Austria
Joost Kok	University of Leiden, Netherlands
Bernd Krämer	FU Hagen, Germany
Liviu Miclea	University of Cluj-Napoca, Romania
Alice Miller	University of Glasgow, UK
Zebo Peng	University of Linkping, Sweden
Alexander Petrenko	ISPRAS, Russia
Mauro Pezzè	University of Milano-Bicocca, Italy
Paolo Prinetto	Politecnico Torino, Italy
Franz Rammig	University of Paderborn, Germany
Konstantinos Sagonas	University of Uppsala, Sweden
Bernhard Steffen	University of Dortmund, Germany
Murat Tanik	University of Alabama, USA
Marcel Verhoef	Chess, Netherlands

Gottfried Vossen	University of Mnster, Germany
Hai Wang	University of Manchester, UK
Alan Wassyng	McMaster University, Canada
Michel Wermelinger	Universidade of Nova de Lisboa, Portugal
Martin Wirsing	LMU München, Germany
Keijiro Yamaguchi	NEC, Japan
Lenore Zuck	NYU, USA

Organization Committee

Andreas Andreou	University of Cyprus - Cyprus
Yannis Demopoulos	University of Cyprus - Cyprus
Chryssis Georgiou	University of Cyprus - Cyprus
Marios Mavronicolas	University of Cyprus - Cyprus

Industrial Board

Mirko Conrad	DaimlerChrysler AG
Limor Fix	Intel, Haifa, Israel (Hardware)
Mike Hinchey	NASA, USA (Space and Robotics)
Manfred Reitenspiess	Fujitsu-Siemens, Munich Germany (Telecommunication Platforms)
Yaron Wolfsthal	IBM, Haifa Israel (HW/SW Systems)
Jianli Xu	Nokia
Yervant Zorian	Virage Logic USA (HW Systems)

Reviewers

Robby
P. Abdulla
E. Abraham
J. Andersson
R. Banach
D. Clarke
J. Deneux
J. Hatcliff
J. Ivers
J. Jacob
D. Kroening
B. Lisper
J. Rehof
M. Schordan
A. Stam
B. Steffen
L. v.d. Torre
A. Wall
Q. Yi
G. Zavattaro
W. Zimmermann

Table of Contents

Interaction and Coordination of Tools for Structured Data

Farhad Arbab[1,2] and Joost N. Kok[2]

[1] Center for Mathematics and Computer Science (CWI)
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
farhad@cwi.nl
[2] Leiden Institute for Advanced Computer Science (LIACS), Leiden University
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
{farhad, joost}@liacs.nl

Abstract. This paper has an introductory nature. It sets the scene for the three papers in the thematic session "Structured Data Tools". We discuss interaction and coordination in general and look more specifically at the XML data format and the dataflow driven coordination language Reo.

1 Introduction

The size, speed, capacity, and price of computers have all dramatically changed in the last half-century. Still more dramatic are the subtle changes of the society's perception of what computers are and what they can, should, and are expected to do. Clearly, this change of perception would not have been possible without the technological advances that reduced the size and price of computers, while increasing their speed and capacity. Nevertheless, the social impact of this change of perception and its feedback influence on the advancement of computer science and technology, are too significant to be regarded as mere by-products of those technological advances.[1]

The social perception of what computers are (to be used for) has evolved through three phases:

1. Computers as fast number crunchers
2. Computers as symbol manipulators
3. Computers as mediators and facilitators of interaction

The general faster-cheaper-smaller trend of computer technology aside, two specific transformations marked the above phase-transitions. The advent of fast, large main memory and mass-storage devices suitable to store and access the significantly more voluminous amounts of data required for non-numerical symbol manipulation made symbolic computation possible. The watershed that set forth the second transition was the availability of affordable computers and digital telecommunication that together fueled the explosion of the Internet.

[1] See [3] for an expanded version of this discussion.

T. Margaria and B. Steffen (Eds.): ISoLA 2004, LNCS 4313, pp. 1–8, 2006.

We are still at the tail-end of the second transition (from symbolic computation to interaction) and trying to come to terms with its full implications on computer science and technology. This involves revisiting some established areas, such as concurrency, from a new perspective, and leads to a specific field of study concerned with theories and models for coordination of interactive concurrent computations. Moreover, it presents new challenges in software engineering to address coarse-grain reuse in component based software and to tackle architectures of complex systems whose organization and composition must dynamically change, e.g., to accommodate mobility, or evolve and be reconfigured to adapt to short- as well as long-term changes in their environment.

Two key concepts emerge as core concerns: *interaction* and *coordination*. While researchers have worked on both individually in the past, we propose that their combination deserves still more serious systematic study because it offers insight into new approaches to coordination of cooperation of interacting components that comprise such complex systems. In our thematic session we concentrate on interaction of components through dataflow. This has two aspects: the nature of data such that they are suited for interaction *and* the coordination of components. First we put interaction and coordination in a context and later we focus on the XML format and the Reo coordination language, being the topics of the papers that belong to this thematic session.

The overview of the rest of the paper is as follows. We discuss interaction and coordination in two separate sections. In these two sections we first treat general issues and then focus on the data aspects. In the final section, we briefly discuss the three papers of the thematic session.

2 Interaction

In the real world, computer systems and databases contain data in incompatible formats. However, interaction implies data exchange between such systems and this exchange has been one of the most time-consuming challenges for developers. Converting the data to a common format can greatly reduce this complexity and create data that can be read by many different types of applications.

Traditional database systems rely on an old model: the relational data model. When it was proposed in the early 1970's by Codd, the relational model generated a revolution in data management. In this model data are represented as relations in first-order structures and queries as first-order logic formulas. It enabled researchers and implementors to separate the logical aspect of the data from its physical implementation. Thirty years of research and development followed, and they led to today's relational database systems.

The advent of the Internet led to a variety of new generation data management applications for which the relational model is no longer suited. For instance, data are now frequently accessed through the Web, is distributed over several sources, and resides there in various formats (e.g., HTML, XML, text files, relational). To accommodate all forms of data and access to them, the database research community has introduced the "semi-structured data model", where data are self-describing, irregular, and graph-like. The new model captures naturally Web data, such as HTML, XML, or other application specific formats like trees and molecules. (See Foundations of Semistructured Data, Dagstuhl Seminar [10].) Data are typically subject to frequent changes in both structure, contents, and interfacing.

As a general trend, data have become more structured: examples of structured data include graphs, trees, molecules and XML documents. In many application areas very large quantities of structured data are generated. Handling these large quantities of structured data requires rigorous data tools. Data tools are tools for handling and expanding the use of such data, including those to acquire, store, organize, archive and analyze data. It is important that these tools are efficient, correct, flexible and reliable. Examples include tools for data conversion, data integration, data security and data mining. Typical areas of application are in health-care, bioscience, financial sector and e-commerce, interoperability and integration.

In our thematic session we focus on XML which is representative of issues in structured data tools. The features of XML that make it particularly appropriate for data transfer are (see [16]):

1. XML uses plain text files.
2. XML can represent records, lists and trees.
3. XML is platform-independent.
4. the XML format is self-documenting in that it describes the structure and field names as well as the syntax for specific values.
5. XML is used as the format for document storage and processing, its hierarchical structure being suitable for most types of documents.
6. XML has already been in use (as SGML) for longer than a decade, and is very popular by itself, with extensive available experience and software.

The XML format is not only a common format, but it also provides a basis for the coordination of systems.

3 Coordination

Coordination languages, models, and systems constitute a recent field of study in programming and software systems, with the goal of finding solutions to the problem of managing the interaction among concurrent programs. Coordination can be defined as the study of the dynamic topologies of interactions, and the construction of protocols to realize such topologies that ensure well-behavedness. Analogous to the way in which topology abstracts away the metric details of geometry and focuses on the invariant properties of (seemingly very different) shapes, coordination abstracts away the details of computation, and focuses on the invariant properties of (seemingly very different) programs. As such, coordination focuses on patterns that specifically deal with interaction.

The inability to deal with the cooperation model of a concurrent application in an explicit form contributes to the difficulty of developing working concurrent applications that contain large numbers of active entities with non-trivial cooperation protocols. In spite of the fact that the implementation of a complex protocol is often the most difficult and error prone part of an application development effort, the end result is typically not recognized as a “commodity” in its own right, because the protocol is only implicit in the behavior of the rest of the concurrent software. This makes maintenance and modification of the cooperation protocols of concurrent applications much more difficult than necessary, and their reuse next to impossible. Coordination languages can be thought of as the linguistic counterpart of the ad hoc platforms that offer middle-ware support for software composition.

Coordination languages are most relevant specifically in the context of open systems, where their participants are not fixed at the outset. Coordination is also relevant in design, development, debugging, maintenance, and reuse of all concurrent systems, where it addresses a number of important software engineering issues. The current interest in constructing applications out of independent software components necessitates specific attention to the so-called *glue-code*. The purpose of the glue-code is to compose a set of components by filling the significant interface gaps that naturally arise among them, simply because they are not (supposed to be) tailor-made to work with one another. Using components, thus, means understanding how they individually interact with their environment, and specifying how they should engage in mutual, cooperative interactions in order for their composition to behave as a coordinated whole. Many of the core issues involved in component composition have already been identified and studied as key concerns in work on coordination. Coordination models and languages address such key issues in Component Based Software Engineering as specification, interaction, and dynamic composition of components. Specifically, *exogenous* coordination models (discussed later in this section) and languages provide a very promising basis for the development of effective glue-code languages because they enable third-party entities to wield coordination control over the interaction behavior of mutually anonymous entities involved in a collaboration activity from outside of those participating entities.

One of the best known coordination languages is Linda [9,13], which is based on the notion of a shared tuple space. The tuple space of Linda is a centrally managed resource and contains all pieces of information that processes wish to communicate with each other. Linda processes can be written in any language augmented with Linda primitives. There are only four primitives provided by Linda, each of which associatively operates on a single tuple in the tuple space. The primitive **in** searches the tuple space for a matching tuple and deletes it; **out** adds a tuple to the tuple space; **read** searches for a matching tuple in the tuple space; and **eval** starts an active tuple (i.e., a process). Numerous other coordination models and language extensions, e.g., JavaSpace of Jini [12,14], are based on Linda-like models.

Besides the “generative tuple space” of Linda, a number of other interesting models have been proposed and used to support coordination languages and systems. Examples include various forms of “parallel multiset rewriting” or “chemical reactions” as in Gamma [6], models with explicit support for coordinators as in Manifold [4,8], “software bus” as in ToolBus [7], and a calculus of generalized-channel composition as in Reo [2]. A significant number of these models and languages are based on a few common notions, such as pattern-based, associative communication [1], to complement the name-oriented, data-based communication of traditional languages for parallel programming. See [15] for a comprehensive survey of coordination models and languages.

Some of the important properties of different coordination languages, models, and systems become clear when we classify them along the following three dimensions: focus of coordination, locus of coordination, and modus of coordination. Although a detailed description of most individual coordination models and languages is beyond the scope of our interest in this paper, an overview of the dimensions of this classification helps to clarify at a more abstract level the issues they address, and thus the concerns of coordination as a field.

Focus of coordination refers to the aspect of the applications that a coordination model, language, or system emphasizes as its primary concern. Significant aspects used by various models as their focus of coordination include data, control, and dataflow, yielding *data-oriented*, *control-oriented*, and *dataflow-oriented* families of coordination models, languages, and systems.

For instance, Linda uses a data-oriented coordination model, whereas Manifold is a control-oriented coordination language. The activity in a data-oriented application tends to center around a substantial shared body of data; the application is essentially concerned with what happens to the data. Examples include database and transaction systems such as banking and airline reservation applications. On the other hand, the activity in a control-oriented application tends to center around processing or flow of control and, often, the very notion of the data, as such, simply does not exist; such an application is essentially described as a collection of activities that genuinely consume their input data, and subsequently produce, remember, and transform “new data” that they generate by themselves. Examples include applications that involve work-flow in organizations, and multi-phase applications where the content, format, and/or modality of information substantially changes from one phase to the next.

Dataflow-oriented models, such as Reo, use the flow of data as the only (or at least the primary) control mechanism. Unlike data-oriented models, dataflow models are oblivious to the actual content, type, or structure of data and are instead concerned with the flow of data from their sources to their destinations. Unlike control-oriented models, events that trigger state transitions are limited to only those that arise out of the flow of data.

Locus of coordination refers to where coordination activity takes place, classifying coordination models and languages as *endogenous* or *exogenous*. Endogenous models and languages, such as Linda, provide primitives that must be incorporated *within* a computation for its coordination. In applications that use such models, primitives that affect the coordination of each module are inside the module itself. In contrast, exogenous models and languages, such as Manifold and Reo, provide primitives that support coordination of entities from *without*.

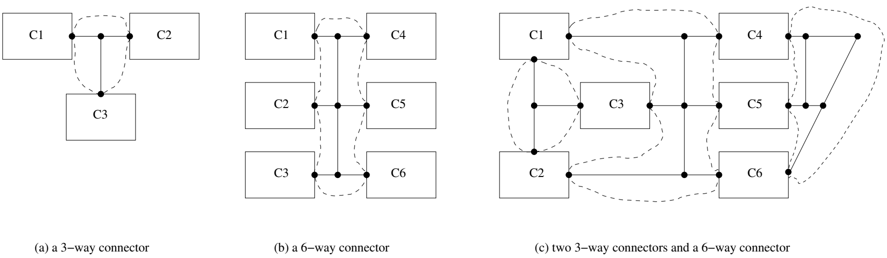
In applications that use exogenous models primitives that affect the coordination of each module are outside the module itself.

Modus of coordination refers to how coordination is carried out in a model or language: how the coordination rules of an application are defined and enforced. The repertoire of coordination rules supported by a coordination model or language can be very different in its nature than that of another. Some, e.g., Linda, Manifold, and Reo, provide primitives for building coordination rules. Others propose rule-based languages where rules act as trigger conditions for action or as constraints on the behavior of active agents to coordinate them in a system. One way or the other, coordination rules provide a level of abstraction which hides much of the complexity of coordination activity from programmers. Models that use more declarative coordination rules can support increased reasoning power.

In our thematic session we consider the Reo language. Reo is a channel-based exogenous coordination model wherein complex coordinators, called *connectors*, are compositionally built out of simpler ones. The simplest connectors in Reo are a set of *channels* with well-defined behavior supplied by users [2]. The emphasis in Reo is on connectors, their behavior, and their composition, not on the entities that connect, communicate, and cooperate through them. The behavior of every connector in Reo imposes a specific coordination pattern on the entities that perform normal I/O operations through that connector, without the knowledge of those entities. This makes Reo a powerful "glue language" for compositional construction of connectors to combine component instances into a software system and exogenously orchestrate their mutual interactions.

Reo's notion of components and connectors is depicted in Figure 1, where component instances are represented as boxes, channels as straight lines, and connectors are delineated by dashed lines. Each connector in Reo is, in turn, constructed compositionally out of simpler connectors, which are ultimately composed out of primitive channels.

For instance, the connector in Figure 1.a may in fact be a flow-regulator (if its three constituent channels are of the right type as described under write-cue regulator, below, and shown in Figure 2.a). Figure 1.a would then represent a system composed out of two *writer* component instances (`C1` and `C3`), plus a *reader* component instance (`C2`), glued together by our flow-regulator connector. Every component instance performs its I/O operations following its own timing



(a) a 3–way connector

(b) a 6–way connector

(c) two 3–way connectors and a 6–way connector

Fig. 1. Connectors and component composition

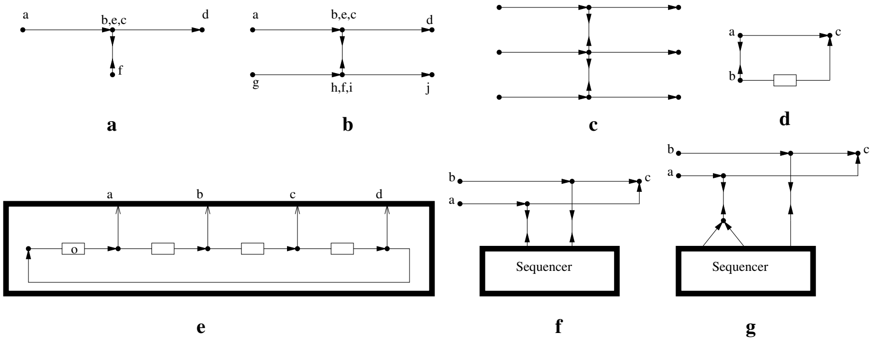


Fig. 2. Examples of connector circuits in Reo

and logic, independently of the others. None of these component instances is aware of the existence of the others, the specific connector used to glue it with the rest, or even of its own role in the composite system. Nevertheless, the protocol imposed by our flow-regulator glue code (see [2] and [5]) ensures that a data item passes from C1 to C2 only whenever C3 writes a data item (whose actual value is ignored): the "tokens" written by C3, thus, serve as cues to regulate the flow of data items from C1 to C2. The behavior of the connector, in turn, is independent of the components it connects: without their knowledge, it imposes a coordination pattern among C1, C2, and C3 that regulates the precise timing and/or the volume of the data items that pass from C1 to C2, according to the timing and/or the volume of tokens produced by C3. The other connectors in Figure 1 implement more complex coordination patterns.

Figure 2.a shows a write-cue regulator connector circuit built out of two synchronous channels and a synchronous-drain, whose behavior is analogous to that of a transistor. Figures 2.b and c show this transistor used to construct more complex Reo connector circuits, specifically, two barrier-synchronizer circuits for, respectively, two- and three-pairs of readers and writers. Figure 2.d shows a Reo connector circuit for an alternator. Figure 2.e shows a sequencer circuit, and Figures 2.f and g show more complex alternator circuits that use this sequencer as a component. See the first paper in this thematic session for a brief overview of Reo and [2] for more details on Reo channels and these and other examples.

There are three papers in the thematic session: the first presents an application of the Reo coordination language; the second paper describes a structured data transformation tool; and the third paper discusses the application of this tool in enterprise architectures. We discuss them in turn.

- *Modeling Coordination in Biological Systems.* The Reo coordination language is used for modeling in systems biology. Various forms of coordination in living cells are discussed and metabolization of galactose is modeled.
- *A Rule Markup Language and its application to UML.* A data transformation tool for XML, called RML, is introduced. RML can be use for rule-based transformations of XML documents. RML rules are stated in XML and are

intended to be mixed with problem-domain specific XML, using the so-called XML wild-card elements.

- *Using XML Transformations for Enterprise Architectures.* An enterprise architecture can be described in an XML document. Using the RML tool, above, different views of this data can be obtained. In the spirit of coordination, the model viewers used are independent of the architectural languages.

We think that these three papers give a good feeling for the type of research in interaction and coordination of tools for structured data. Such tools are needed for computers as mediators and facilitators of interaction.

References

1. J.-M. Andreoli, P. Ciancarini, and R. Pareschi. Interaction Abstract Machines. In *Trends in Object-Based Concurrent Computing*, pages 257–280. MIT Press, 1993.
2. F. Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, June 2004.
3. F. Arbab. Coordination of interacting concurrent computations. In *[11]*. 2005.
4. F. Arbab, I. Herman, and P. Spilling. An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, 5(1):23–70, February 1993.
5. F. Arbab and J.J.M.M. Rutten. A coinductive calculus of component connectors. In D. Pattinson M. Wirsing and R. Hennicker, editors, *Recent Trends in Algebraic Development Techniques, Proceedings of 16th International Workshop on Algebraic Development Techniques (WADT 2002)*, volume 2755 of *Lecture Notes in Computer Science*, pages 35–56. Springer-Verlag, 2003.
6. J.-P. Banâtre and D. Le Métayer. Programming by multiset transformations. *Communications of the ACM*, 36(1):98–111, January 1993.
7. J. Bergstra and P. Klint. The ToolBus Coordination Architecture. In P. Ciancarini and C. Hankin, editors, *Proc. 1st Int. Conf. on Coordination Models and Languages*, volume 1061 of *Lecture Notes in Computer Science*, pages 75–88, Cesena, Italy, April 1996. Springer-Verlag, Berlin.
8. M.M. Bonsangue, F. Arbab, J.W. de Bakker, J.J.M.M. Rutten, A. Scutellá, and G. Zavattaro. A transition system semantics for the control-driven coordination language Manifold. *Theoretical Computer Science*, 240:3–47, 2000.
9. N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.
10. Dagstuhl. http://www.dagstuhl.de/05061/.
11. D. Goldin, S. Smolka, and P. Wegner, editors. *Interactive Computation: The New Paradigm.* Springer-Verlag, 2005. (to appear).
12. Jini. http://www.sun.com/jini.
13. W. Leler. Linda meets Unix. *IEEE Computer*, 23:43–54, February 1990.
14. S. Oaks and H. Wong. *Jini in a Nutshell.* O'Reilly & Associates, 2000.
15. G.A. Papadopoulos and F. Arbab. Coordination models and languages. In M. Zelkowitz, editor, *Advances in Computers – The Engineering of Large Systems*, volume 46, pages 329–400. Academic Press, 1998.
16. XML. http://www.w3schools.com/xml.

Modelling Coordination in Biological Systems

Dave Clarke, David Costa, and Farhad Arbab

CWI
PO Box 94079, 1090 GB Amsterdam
The Netherlands
{dave, costa, farhad}@cwi.nl

Abstract. We present an application of the *Reo* coordination paradigm to provide a compositional formal model for describing and reasoning about the behaviour of biological systems, such as regulatory gene networks. *Reo* governs the interaction and flow of data between components by allowing the construction of connector circuits which have a precise formal semantics. When applied to systems biology, the result is a graphical model, which is comprehensible, mathematically precise, and flexible.

1 Introduction

Within a biological system, complex *biological pathways*, the interconnection between genes and proteins and other chemical reactions within organisms, form the basic fabric of life. Systems biology aims to integrate information about biological entities and their relationships with the aim of understanding the complex metabolic networks and the rôle of genes within them. Beginning with incomplete information about a system, biologists produce a mathematical model of their system, which is ultimately used to predict the behaviour of the system. By testing their model against experimental data, biologists can produce successively more accurate models. But as biologists study larger systems, their modelling technology is proving to be insufficient. What systems biologists need are formal techniques to describe reaction networks, giving, for example, an algebraic description of their behaviour, enabling abstraction from molecular actions, if desired. The models should enable composition of larger models from smaller ones, and tools should be provided for analysing the behaviour of these models under a variety of boundary conditions [1]. A number of models have recently been proposed to fill this gap. Frequently, these models stem from the study of concurrency theory, which traditionally provides theoretical foundations for concurrent and distributed computer systems. Some models are based on process calculi [2,3,4,5,6,7,8], whereas others adopt circuit or network-based formalisms [9,10,11,12]. An immediate advantage of these approaches to modelling is that the extensive theory that exists for reasoning about concurrency can be applied to biological systems (for example [13]). Furthermore, by providing formally precise, executable models of biological systems, these approaches represent a promising path toward understanding highly complex biological systems.

An alternative approach is to use a framework which abstracts away from the behaviour of agents and focus on their interaction both at a primitive level

T. Margaria and B. Steffen (Eds.): ISoLA 2004, LNCS 4313, pp. 9–25, 2006.

and across the entire system. Understanding biological systems requires understanding, for example, the gene networks which *regulate* protein production through the *activation* and *suppression* of various enzymes and other genes. Often mere *competition* for molecules is the means by which nature achieves organisation through distributed *control* [1]. In their most general setting, regulation, activation, suppression, competition, and control are studied within the inter-disciplinary field *Coordination Theory* [14].

The *Reo* coordination model for coordinating software components [15] also falls within this general theory. Rather than focusing on what components (or processes) do in isolation, *coordination models*, including *Reo*, focus on the composition of components and their interaction, generally by governing the *flow* of data between components. The coordination layer prescribes/describes the interaction between the components. This change of focus enables one to understand a system at a higher level of abstraction, focusing on the interaction and control aspects rather than on the entities being controlled. It is from this perspective that we endevour to contruct models for use in systems biology. *Reo* coordinates components using connectors composed of primitive channels. These connectors can be seen as circuits that capture the interaction, data flow, synchronisation, and feedback among components. Circuits have a graphical representation and precise semantics [16,17], and allow the composition of models of large systems out of smaller ones. This paper applies *Reo* to modelling biological systems.

2 Evidence of Coordination in Biological Systems

In biological systems the presence of coordination mechanisms is evident at different levels and in different ways [18,14,19,20]. In general, coordination takes place in a highly distributed manner, but we can break it down into a number of categories: inter-cellular coordination, boundary coordination, intra-cellular coordination, and gene coordination.

Inter-cellular coordination. A cell is coordinated via interaction with its environment. This can include interaction with other cells. All cells receive and respond to signals from their surroundings. The simplest bacteria sense and swim toward high concentrations of nutrients, such as glucose or amino-acids. Many unicellular eukaryotes also respond to signalling molecules secreted by other cells, allowing cell-cell communication. It is, however, in multi-cellular organisms where cell-cell communication reaches its highest level of sophistication. The behaviour of each individual cell in multi-cellular plants and animals must be carefully regulated to meet the needs of the organism as a whole. The function of the many individual cells in a multi-cellular organism is integrated and coordinated via a variety of signalling molecules that are secreted on the surface of one cell and bound to a receptor present on another cell.

Boundary coordination. A cell's internal behaviour is stimulated or, more generally, regulated via interaction on its boundary. Most cell surface receptors stimulate target enzymes which may be either directly linked or indirectly coupled to receptors. A chain of reactions transmits signals from the cell surface to a variety of intra-cellular targets. The targets of such signalling pathways frequently include factors that regulate gene expression. Intra-cellular signalling pathways thus connect the cell surface to the nucleus, leading to changes in gene expression—the internal coordinator of cell behaviour—in response to extra-cellular stimuli. Changes in expression lead to different metabolic pathways.

Intra-cellular coordination. Intra-cellular reactions regulate all aspects of cell behaviour including metabolism, movement, proliferation, survival and differentiation. Metabolism is a highly integrated process. It includes *catabolism*, in which the cell breaks down complex molecules to produce energy, *anabolism*, where the cell uses energy to construct complex molecules and perform other biological functions, and more general *metabolic pathways* consisting of a series of nested and cascaded feedback loops which accommodate flexibility and adaptation to changing environmental conditions and demands. Metabolism is regulated through competition for resources, and by positive and negative feedback. Negative feedback (usually by end-product inhibition) prevents the over-accumulation of intermediate metabolites and contributes to maintaining homeostasis—a system's natural desire for equilibrium.

Gene coordination. Gene behaviour plays the most significant coodination rôle, ultimately being the central coordination mechanism within the entire cell. *Gene Regulatory Networks* are a model which represents and emphasizes the genes' rôle in all activities within a cell. Genes regulate and orchestrate the different phases that comprise a metabolic pathway. A gene regulatory network can be understood as a complex signalling network in which all coordination between different cell entities is dictated dynamically by the genes directly, or indirectly through coordination mechanisms such as *suppression/negative regulation* and *activation/positive regulation.* Furthermore, gene behaviour is itself regulated via the products of metabolism and via self-regulatory feedback.

3 Coordination in *Reo*

Coordination models and languages enable the control of the interaction behaviour of mutually anonymous components (or processes) from outside of those components. Rather than allowing components to communicate directly, a coordination model intervenes to regulate, inhibit, and direct the communication and cooperation of independent components. *Reo* is a powerful channel-based coordination model wherein complex coordinators, called *connectors*, are compositionally built out of smaller ones. Every connector in *Reo* imposes a specific coordination pattern on the entities that interact via the connector [15].

The most primitive connectors in *Reo* are *channels.* A channel has exactly two ends, each of which may be an *input* end, through which data enter, or an *output* end, through which data leave the channel. Channels may have an input end and an output end, or two input ends, or even two output ends. *Reo* places no restriction on the behaviour of channels, as long as they support certain primitive operations such as I/O. This allows an open set of different channel types to be used simultaneously together in *Reo*, each with its own policy for synchronisation, buffering, ordering, computation, data retention/loss, etc. A number of basic channels are [15]:

Sync. A synchronous channel is denoted ⟶. A data item is transmitted through this channel when both a write on its input end and a take on its output end are pending.

LossySync. A lossy synchronous channel is denoted - - - - ▸. This channel behaves like a synchronous channel whenever a take on the output end is pending. However, if a write is performed to its input end and no take is pending, the input is performed, but data is lost and thus not transfered.

SyncDrain. A synchronous drain is denoted →——←. A data item flows only when writes are pending on each of its two input ends. The effect is to synchronise the two writers. The data is lost.

SyncSpout. A synchronous spout is denoted ←—→. A data item flows only when takes are pending on each of its two output ends. The effect is to synchronise the two readers.

FIFO1. A FIFO1 channel has buffer of size one. A write to its input end can succeed only if the buffer is empty, after which the written value is stored in the buffer. Otherwise the writer blocks. A take from its output end only succeeds if the buffer is full. A FIFO1 which is initially empty is denoted —▭—→, and one which is initially full is denoted —[0]—→.

A *Reo connector* is a set of channel ends and their connecting channels organized in a graph of *nodes* and edges such that: zero or more channel ends coincide on every node; every end coincides on exactly one node; and there is an edge between two (not necessarily distinct) nodes if and only if there is a channel whose two ends coincides on each of the nodes. The coincidence of channel ends at a node has specific implications on the data flow among and through those ends. There are three kinds of node, depending upon the kinds of coincident ends. *Input nodes* have only input ends, *output nodes* have only output ends, and *mixed nodes* have at least one of each. A component connected to an input node may write to that node, which then acts as a *replicator*, copying the data item to all ends coincident on the node. Similarly, a component connected to an output node can take data if data is available on any one of the output ends coincident on the node. Thus the node acts as a non-deterministic *merger* of the coincident output ends. A mixed node acts like a self-contained pumping station, combining the behaviour of an output node (merger) and an input node (replicator), by non-deterministically taking a suitable data item offered by any one of its coincident output ends and replicating it to all of its coincident input ends. This operation succeeds only when all the output ends attached to the node are able to accept the data. Graphically nodes are represented using "•".

The last feature of *Reo* we mention is *encapsulation*. This enables abstraction of the details of a connector. It it representated as a box enclosing a circuit.

We now present an example of a *Reo* connector to illustrate the complex behaviour which can emerge through composition of simple channels. An *exclusive* *router* is depicted in Figure 1, along with the shorthand mnemonic "⊗" which we will use subsequently to represent the instances of this connector. Each data item entering via node A will be synchronously passed to either node B or node C, but not both, depending upon which of B and C first makes a request for data. Ties are broken non-deterministically [16]. This behaviour emerges in an non-obvious manner simply by composing together a few simple channels.

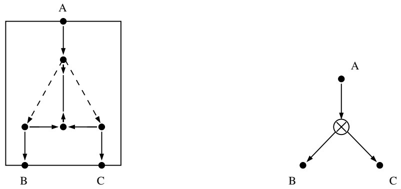
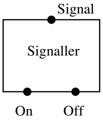


Fig. 1. (a) Exclusive Router connector and (b) the Mnemonic for its instances

Another example, used in our modelling, is a *signaller*. A signaller, see below, can alternate between two different states, *On* or *Off*. When the signaller is in the *On* state, an output can be emitted on the **Signal** channel end. In the *Off* state, no output is emitted on the **Signal** channel end. The state of a signaller is dictated by inputs on the channel ends labelled **On** and **Off**. An input on **On** switches the signaller to the *On* state, if the state of the signaller is *Off*, or is otherwise ignored. Similarly, an input on **Off** switches the signaller to the state *Off*, or is ignored. A signaller, with initial state *Off*, is represented using the following mnemonic:



If the connection to either the **On** or **Off** end of a signaller is not known, we omit the end from the diagram.

We now present the full *Reo* circuit implementing the signaller in terms of primitive channels. This demonstrates the expressiveness of *Reo*. An alternative is to supply a signaller as a primitive with its behaviour defined directly using constraint automata.

Sequencer. The sequencer connector [15] (for 2 elements) is depicted in Figure 2. This connector is used to alternate the behaviour at nodes **A** and **B**, given by the regular expression: $(\mathbf{AB})^*$. The implementation of the sequencer consists of a loop of *FIFO1* buffers, of which all but one are empty, which implements a token ring, enabling **A** and **B** to input data alternately. The token is used to indicate which synchronous channels may pass data. After **A** inputs data, the token moves to the other *FIFO1* buffer, enabling the **B** to input data. After **B** inputs, the sequencer returns to its initial state, accepting **A**.

SwitchConverter. The token passing loop which forms a part of the sequencer can readily be adapted to act as the core of a switch that enables or inhibits the flow of data, toggled by alternating values from a third channel end (see the PreValve circuit, below). A more natural switch consists of two separate channel ends, one corresponding to *On*, the other corresponding to *Off* (repeated inputs in **On** or **Off** channel ends have no effect). The SwitchConverter connector (Figure 2) converts the latter kind of switch into the former (conversion in the other direction is also possible).

PreValve. The PreValve connector is depicted in Figure 2. Its initial state can be either *On* or *Off*. When the connector is in the *On* state, data can be inputed continually through the **Flow** channel end. In the *Off* state, not data can be inputed. Data on the **Toggle** channel end toggles the connector between its *On* and *Off* states. Note that, this circuit is the core of the Valve connector in [16].

Signaller. The detailed implementation of this circuit is given in Figure 4.

4 Case Study: Galactose Utilization in Yeast

When a biologist wants to understand a system, such as the rôle of various genes in regulating a metabolic pathway, they construct a model of the system. Initially, the model is a black box. Through successive refinements, based on experimental data, more accurate models are developed. The ultimate goal of this process is to find models which are accurate enough to be used in a predictive manner, giving an indication of how the system being modelled will behave under conditions outside those covered by existing experimental data.

In this section, we outline an approach to developing models using *Reo*, and apply it in a case study, the biological system *Galactose Utilization in Yeast* [21]. In particular, we focus on the process of metabolizing *Galatose* to *Glucose-6-P* (Figure 5). The system's behaviour can be described as following [21]:

> Yeast metabolizes galactose through a series of steps involving the *GAL2* transporter and enzymes produced by *GAL1*, *7*, *10*, and *5*. These genes are transcriptionally regulated by a mechansim consisting primarily of *GAL4*, *80*, and *3*. *GAL6* produces another regulatory factor thought to repress the GAL enzymes in a manner similar to *GAL80*.

In order to produce a model, we need to determine the level of abstraction at which the model will work. The finite set of *biological entities* (or just *entity*) that play a rôle in a biological system (or process) is called its *domain*. Entities can be any cell organelle (e.g., mitochondrion, ribossome), suborganelle constituent (e.g., gene), cell component (e.g., membrane), chemical, etc. For the present case study, the domain consists of *seven chemical substances* (*Gal-out*, *Gal-in*, *Gal-1-P*, *UDP-Glu*, *UDP-Gal*, *Glu-1-P*, *Glu-6-P*); *five chemical reactions* (*Gal-out→Gal-in* (actually a membrane crossing), *Gal-in→Gal-1-P*, *Gal-1-P+UDP-Glu→Glu-1-P+UDP-Gal*, *UDP-Gal→UDP-Glu*, *Glu-1-P→Glu-6-P*); and *nine genes* (*Gal1*, *Gal2*, *Gal3*, *Gal4*, *Gal5*, *Gal6*, *Gal7*, *Gal10*, *Gal80*).

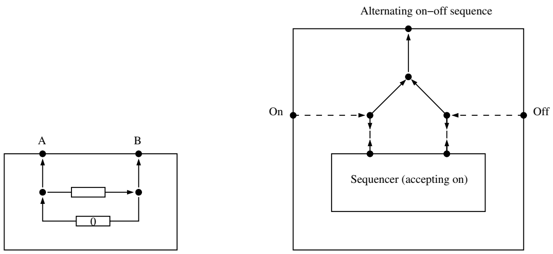


Fig. 2. (a) Sequencer connector (**A** is enabled) (b) Switch Converter connector

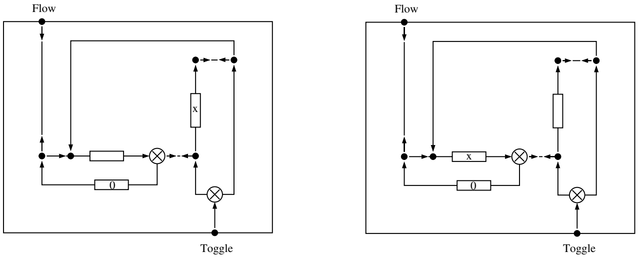


Fig. 3. PreValve connector – State On and State Off

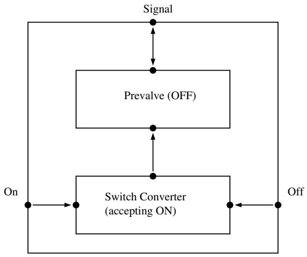


Fig. 4. Signaller connector (Initial state is *Off*)

The next step in the modelling process is to instantiate each domain element with a suitable *Reo* connector. This may be as simple as selecting the appropriate connector from a library, or it may require a new model to be developed.

Genes as Signallers. The first step in modelling the behaviour of a gene as a *Reo* connector is to extract the relevant characteristics and behavioural properties that genes express in the biological system. Once this behaviour has been determined, up to the chosen level of abstraction, a *Reo* connector can be constructed to match that behaviour.

Genes are segments of DNA within chromosomes which cells transcribe into RNAs and translate, at least in part, into proteins. Genes affect behaviour within the cell through the proteins which are produced. Some of the resulting proteins

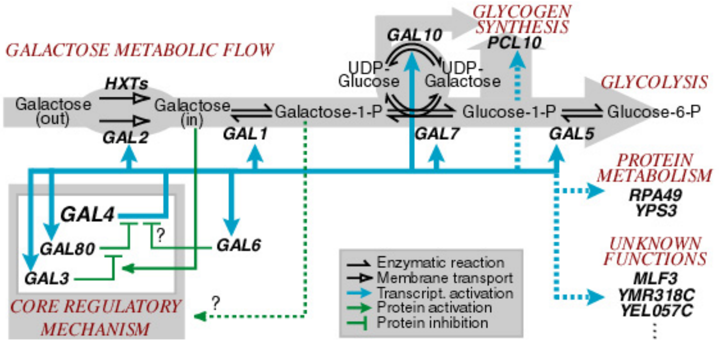


Fig. 5. The Galactose System. The top gray section depicts the chemical reactions involved. The bottom part depicts the gene regulatory network coordinating the reactions. From [21].

are enzymes which may regulate may catalyze a chemical reaction. Other proteins are transcription factors which may regulate gene behaviour—though for modelling purposes, these indirections are secondary issues.

We can abstract this behaviour using a *signaller Reo* connector, described in Section 3. The nine genes can thus be modelled by *signaller* connectors.[1]

Chemicals. The five chemical substances that either act as reactants or products of reaction are modelled as *Reo* nodes. More precisely, we introduce a node into a connector and associate "data flowing through the node" with "the presence of the chemical." As we outline in Section 5, such nodes can be used to make observations about these entities within the system.

Modelling Chemical Reactions. A chemical reaction can either be the synthesis of a substance from two or more individual molecules coming together, or the decomposition of a molecule into smaller molecules. A reaction may have multiple reactants and multiple products, and may require the presence of an enzyme.

A chemical reaction can be modelled simply as a number of input channel ends (one for each reactant), a number of output channel ends (one for each product) and an input for the enzyme signal required to enable the reaction. A signal on one of the reactants' channel ends signals the availablitiy of a reactant. The reaction proceeds whenever a signal is available on all reactants' channel ends and the enzyme channel end. This simultaneous availability is imposed using a *SynchDrain* channel. A *FIFO1* channel is placed in the circuit after the reaction has occured to model the delay present in a chemical reaction.

Four chemical reactions (*Gal-out*→*Gal-in*, *Gal-in*→*Gal-1-P*, *UDP-Gal*→ *UDP-Glu*, *Glu-1-P*→*Glu-6-P*) need one reactant and output one product. The

[1] An **enzyme** is a protein whose presence catalyzes (speed up) chemical reactions without being itself consumed. It can be modelled using a *signaller*. For simplicity, we have folded enzyme behaviour into that of the gene which produces it.

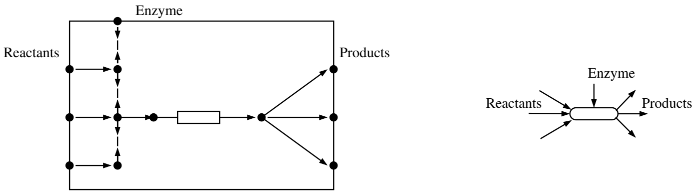


Fig. 6. (a) Connector for Complex Reactions with Enzyme and (b) its Mnemonic

remaining reaction (*Gal-1-P*+ *UDP-Glu*→*Glu-1-P*+ *UDP-Gal*) needs two reactants and outputs two products. These are both modelled using variants of the connector in Figure 6.

Now that we have a domain and the behaviour of the entities which it comprises, we can start to build a *Reo* model. To do so, we must ask a second question: how de we construct models of complex systems which comprise multiple biological entities?

We begin by determining the relationship between the entities which describes, for example, that one entity activates, or more generally, regulates, another entity. This also includes basic connections between chemical reactions where the product of one entity is a reactant of another. We also identify *boundary entities* whose behaviour does not depend on and which is not regulated by other entities that we are modelling. For our example, the boundary entities include two elements *Gal-out* and *Glu-6-P*. Entities which are not on the boundary are considered to be *internal*, and their behaviour typically depends on or is regulated entirely by the activity of other entities in the model.

Each of the interactions between entities must be modelled in *Reo*. Usually, this consists simply of composing the connectors involved in the right manner. In our model, we determine the relationship between entities using Figure 5.

Composition: Activation/Presence, Suppression/Absence. The action of a gene is to provide the proteins vital to a cell. An enzyme catalyzes chemical reactions, and transcription factors regulate (activate or suppress) gene activity. The behavior of activation and suppression can be modelled by composing the signaller connector which models the gene, enzyme, or transcription factor with the circuit modelling the entity which it regulates. To model activation/presence (or

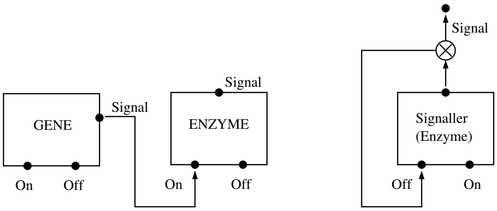


Fig. 7. (a) A Gene activates an Enzyme. (b) Self-Regulation using Feedback

suppression/absence) by one signaller on some target signaller, the **Signal** channel end of the first signaller is connected to the **On** (or **Off**) channel end of the target signaller, as depicted in Figure 7(a). Multiple sources of activation or suppression present in a system can simply be modelled by merging signals from various sources via a *Reo* node.

Composition: Self-regulation. Self-regulation means that a biological entity directly or indirectly is regulated by itself. Products of the biological process influence, either positively or negatively, the process which created those products. To model self-regulation, we use feedback in a *Reo* circuit. A simple example is presented in Figure 7(b). This figure models a gene whose products (the signal) cause it to turn itself off. For more complex situations where the regulation depends upon time, reaction rates, or concentration, may require timed *Reo* circuits to be more accurately modelled [22].

Composition: Reaction Pathways, Reversible Reactions. Composition permits the building of chains of chemical reactions, in which the product of one reaction is the reactant of another. Reaction chains are the fundamental elements of biological pathways, as we discussed in Section 2. Reversible reactions can also be modelled by suitably composing the forward and reverse reactions together. The reactants involved can either become reactants in the reaction going in the other direction, or be used by another reaction in the pathway.

The Composed System. We can now compose all of our entities together. Figure 5 informs whether the regulation from a gene is positive or negative. This information is used to determine how to connect genes together, by connecting a postive factor to the **On** port of the entity it regulates positively, and so on, as

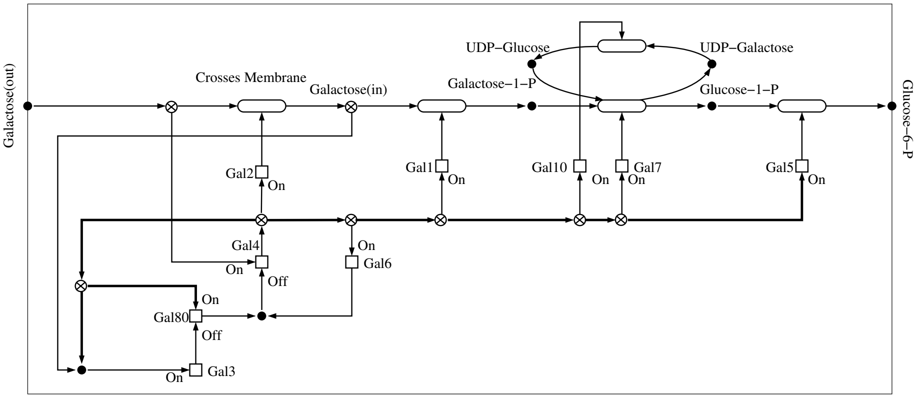


Fig. 8. Galactose Metabolism modelled using *Reo*
Boxes denote signallers—unknown activation/suppression relationships are omitted. Ovals denote chemical reactions. The significant rôle *Gal4* plays is also highlighted. Exclusive routers ⊗ model that each signal is used once.

outlined above. Finally, the appropriate gene action is connected to each chemical reaction, and these reactions are chained together. The result is the circuit in Figure 8.

5 Reasoning About *Reo* Models

Having constructed a *Reo* model, we need techniques for reasoning about its behaviour. This will include checking whether the model fits the experimental data, and determining more general properties, such as the presence of stable states or cycles of states [23] within the model, and understanding the relationships (both causality and cooperation) between various entities. Our model permits reasoning about the relationship between the input and output behaviour of boundary entities. Various behaviours can be determined by *perturbing the boundary*—by setting up different boundary conditions, the behaviour at the remainder of the boundary can be determined. The idea can easily be extended to model internal behaviour also, simply by exposing the internal entities of the model on the boundary.

Reo semantics have been defined in terms of two different formalisms: (timed) constraint automata[2] and abstract behaviour types (not used here) [16,17]. In addition, a number of modal logics and model checking algorithms have been developed for specifying and checking properties of *Reo* circuits. This machinery can be readily applied to biological models.

Constraint Automata. A constraint automaton [17] is an automaton which describes the sequence of possible observations on the boundary nodes of a *Reo* connector. Such an automaton is defined over a set of names $\mathcal{N} = \{A_1, \ldots, A_n\}$, which correspond to the input/output nodes. We present only "data-insensitive" automata. A constraint automaton is a tuple $\mathcal{A} = (\mathcal{Q}, \mathcal{N}, \longrightarrow, \mathcal{Q}_0)$, where $\mathcal{Q}$ is a finite set of states; $\mathcal{N}$ is a finite set of names; $\longrightarrow$ is a finite subset of $\mathcal{Q} \times 2^{\mathcal{N}} \times \mathcal{Q}$, called the transition relation of $\mathcal{A}$, written $q \xrightarrow{N} p$, with the constraint that $N \neq \emptyset$; and $\mathcal{Q}_0 \subseteq \mathcal{Q}$ is the set of initial states.

An automaton starts in an initial state $q_0 \in Q_0$. If the current state is q, then the automaton waits until signals occur on some of the nodes $A_i \in N$. If signals are observed at nodes A_1 and A_2, for example, and at no other nodes at the same time, then the automaton may take a transition $q \xrightarrow{\{A_1, A_2\}} p$. A *run* of an automaton is a sequence of non-empty subsets of $\mathcal{N}$, $(N_0, N_1, N_2, \ldots)$ which corresponds to a series of signals occuring simultaneously at the nodes of the *Reo* connector which the automaton models.

There are two important constructions on constraint automaton. The first construction, *product*, denoted $\mathcal{A} \bowtie \mathcal{B}$, takes two constraint automata and produces an automaton which is the result of "joining" the actions on names shared between the two automata. This captures the composition of *Reo* circuits, though

[2] Timed constraint automata play no role in the models presented in this paper. We anticipate that time will play be required to produce more refined models.

the details are too involved to go into here. This operation is similar in operation to the join operation on relational databases [17].

The second construction, *hiding*, denoted $\exists[C]\mathcal{A}$, takes a name, C, and a constraint automaton, $\mathcal{A}$, and produces an automaton where all behaviour at node C is internalised. Observations about C are no longer possible. Paths involving just label $\{C\}$ are compressed to eliminate empty paths in the resulting automaton. The resulting automaton has the same behaviour on the other nodes.

Projection to Subsystem of Interest. The first step when reasoning about a *Reo* circuit is to construct a constraint automaton for it by constructing the product of the automata which model its channels and nodes. Assume now that the resulting automaton, $\mathcal{A}$, has nodes $\{A, B, C, D, E\}$, where $\{A, B, C\}$ are on the boundary and $\{D, E\}$ are internal. At this stage, the constraint automaton contains all the information which occurs on every node. Normally, hiding would be used to produce an automaton which models the behaviour on the boundary of the connector, i.e., $\exists[D, E]\mathcal{A}$. However, the fact that the automaton resulting from a product alone keeps the behaviour of the internal nodes exposed means that we can use the automaton to reason about the internal behaviour, simply by not hiding them. Thus if a different set of nodes is of interest to the reasoner, an different automaton can be produced containing only those nodes. For example, the automaton $\exists[A, B, E]\mathcal{A}$ can be used to understand the relationship between boundary node C and internal node D.

Causality and Cooperation Analysis. A temporal logic called Time Scheduled-Data-Stream Logic (TSDSL) has been defined, along with model checking algorithms, for the timed version of constraint automata [22]. Scheduled-stream logic (SSL) is the obvious simplification of TSDSL, removing references to time and data. Its formulae over a node set $\mathcal{N}$ are given by the grammar:

$$\phi ::= \mathsf{true} \mid \phi_1 \wedge \phi_2 \mid \neg\psi \mid \langle\!\langle \alpha \rangle\!\rangle \psi \mid \psi_1 \,\mathsf{U}\, \psi_2$$
$$\alpha ::= N \mid \alpha_1 \vee \alpha_2 \mid \alpha_1 \wedge \alpha_2 \mid \alpha_1 ; \alpha_2 \mid \alpha^*$$

N is a nonempty subset of $\mathcal{N}$. α is a so-called schedule expression, giving a regular expression for finite sequences of subsets of $\mathcal{N}$, corresponding to a sequence of "events". The formula $\langle\!\langle \alpha \rangle\!\rangle \psi$ states that each run has a prefix in the set described by α, with the suffix of the run satisfying ψ. Lastly, $\psi_1 \,\mathsf{U}\, \psi_2$ is the until modality from Linear Temporal Logic [24], stating that ψ_1 must hold up until the particular point which ψ_2 holds. This logic can express properties of runs of an automaton.

Biologists wish to pose a number of questions about a model. These often take the general form, *what happens if I press this button*? More concretely, they ask: what is the cause of gene *Gal80* being on? Does *Gal80* affect product *Galactose-1-P*? Do *Gal10* and *Gal7* cooperate to produce *Glucose-1-P*? Is this cooperation necessary? Can other entities produce *Glucose-1-P*? Can *Glucose-1-P* be produced without *Gal10*? Given a constraint automata at the appropriate level of abstraction, with nodes hidden to avoid "noise", such questions can be expressed in SSL as assertions over the visible node set. The validity of the assertions can be determined using model checking. Now if, $[\![\alpha]\!]\psi \equiv \neg\langle\!\langle\alpha\rangle\!\rangle\neg\psi$, $\mathsf{false} \equiv \neg\mathsf{true}$, and $\langle\!\langle\neg Gal4\rangle\!\rangle\psi \equiv \langle\!\langle N_1 \vee N_2 \vee \cdots\rangle\!\rangle\psi$, where each N_i is a subset of nodes that do not include *Gal4*, we can readily make assertions such as:

- $\langle\!\langle Gal4\rangle\!\rangle\mathsf{true}$ – *Gal4* must signal;
- $[\![Gal4]\!]\mathsf{false}$ – *Gal4* cannot signal;
- $[\![Gal80]\!]([\![Gal4]\!]\mathsf{false}\,\mathsf{U}\langle\!\langle \textit{Gal-out}\rangle\!\rangle\mathsf{true})$ – after *Gal80* has signalled, it is not possible for *Gal4* to signal until *Galactose* has been detected on the outside of the cell;
- $[\![Gal4]\!]\langle\!\langle Gal80\rangle\!\rangle\mathsf{true}$ — *Gal4* turns *Gal80* on;
- $[\![Gal80]\!]\mathsf{false} \Rightarrow \langle\!\langle\neg Gal4\rangle\!\rangle[\![Gal80]\!]\mathsf{false}$ — only *Gal4* turns *Gal80* on;
- $[\![\{\textit{Gal-in}, Gal1\}]\!]\langle\!\langle \textit{Gal-1-P}\rangle\!\rangle\mathsf{true}$ — *Galactose* detected inside the cell and the signalling of *Gal1* together are required to produce *Galactose-1-P*; and
- $[\![Gal6 \vee Gal80]\!][\![Gal4]\!]\mathsf{false}$ — the presence of either *Gal6* or *Gal80* can stop *Gal4* from signalling.

Many other questions regarding the states of a system and the pathways therein can be formulated [13].

Generating Experiments from Models. An important reason for having accurate models is to understand the cause and effects of disease and the effects of drugs used to treat them. Models of biological systems will typically, at least initially, be insufficiently accurate for the desired purpose. In order to refine a model, a biologist will need hypotheses to test experimentally. To generate experiments to test the behaviour of disease and drug treatment, in the case where they act at a genetic level, the behaviour of the faulty gene or drug needs to be modelled.

The first approach is to extract the consequent behaviour from an existing *Reo* model. This is done by first characterising the behaviour of the faulty gene or drug as streams of observations at some visible node in the model. The constraint automaton can then be reduced to one which contains only the streams of observations selected by the biologist, and can then be analysed using the techniques above to generate new hypotheses. This approach, however, is only capable of producing behaviour which is already present in the model.

The second approach is to modify the *Reo* model by replacing the connector corresponding to the entity of interest by a new connector which captures the behaviour of the gene malfunction or the action of the drug introduced into the system. The result is a new *Reo* model with the desired behaviour built-in.

In both cases, biologists can study a model by subjecting it to the reasoning techniques described above, and use it to generate new hypotheses. An advantage of using the first technique, in the event that the experimental data matches the generated hypothesis, is that the behaviour is already included within the model, thus no change to the model need be made. Otherwise, if either the experimental data and the model do not match, or the second technique is used, appropriate refinement of the model will be required.

Simulation. Another technique available to biologists is simulation. We have developed tools for simulating *Reo* circuits and constraint automata. These tools are applicable to simulating biological systems.

Unsoundness, Incompleteness and Refinement. Modelling is an iterative process whereby a model is refined, based on experimental data produced by a biologist, to produce a more accurate model. A model can be inaccurate in two ways: it can be unsound or incomplete. Soundness is the requirement that any behaviour which the model exhibits is also exhibited by the biological system. Completeness is the other way around, namely, that any behaviour observed in the real system can be reproduced in the model. (These definitions are only relative to the level of abstraction at which the modeller is operating.) Unsound (*false positives*) and incomplete (*false negatives*) behaviour can be discovered by both probing the model, and by testing hypotheses experimentally in the biological system, and by comparing simulations against experimental results. When the predictions of the model fail to match the results of the experiment, the model needs to be refined, that is, to be adapted to preserve all previous sound behaviour and also capture (or exclude) the new experimental observations. The refinement technique is related to software evolution in the presence of changing specifications. We do not yet have a clear approach to using the "evidence" of unsoundness or incompleteness in one model to produce a model which is more sound and complete.

6 Related Work

A number of researchers have expressed the need for larger scale models of biological systems [1,21,25]. A number of different modelling approaches have emerged to meet this need. Many depart dramatically from the traditional, small-scale approaches based on differential equations and Monte Carlo simulation. In this section, we give a taste of what these new approaches are.

A number of biological modeling languages are based on process calculi. The pi-calculus has been used for modeling general reaction pathways [2], variations of the ambient calculus have modeled systems involving membrane interactions [7,26,5], and special purpose calculi have been used to model protein-protein interactions [4]. In some cases, a stochastic element is added, making the models surprisingly accurate [27]. In general, this approach displays a lot of flexibility, tools for reasoning about the models exist or can be readily adapted from existing tools [13], and process calculi can be simulated.

Boolean Networks are one of the first models of gene regulatory network behaviour [23]. These consist of a number of boolean states, corresponding to whether genes are active or not, and a transition function which *lock-step* determines the next state. This process is iterated, and the network can be analysed for stable cycles of states, called *attractors*. These models have the advantages of simplicity and that they are readily simulated, but they are limited by their discrete nature and their lock-step evaluation. Some of the limitations have been overcome, by introducing probability to the model [28]. More advanced network models break away from the lock-step evaluation of Boolean networks, by incorporating continuous aspects, producing hybrid models which have both discrete and continuous factors. Such models include the circuits of McAdams and Shapiro [12] and models based on hybrid petri nets [11], and hybrid automata [29]. These models capture direct information flow within a biological system, and computational techniques often exist to determine indirect relationships and effects. Although more flexible than Boolean networks, more computational effort is required to analyse these models. Our approach sits somewhere between these two, not being restricted to lock-step evaluation and being relatively simple to analyze.

Coordination is a buzzword commonly used when talking about (systems) biology [18], although, to our knowledge, this paper is the first attempt at applying a coordination model in this area. We expect to offer significant advantages because coordination, prevalent in Biology, is foremost in our model. Compared with approaches based on other conconcurrent formalisms, our model often works at a higher level of abstraction, because the coordination (synchronisation among multiple entities, for example) needs to be programmed in a process calculus model, whereas in *Reo* it comes for free, either directly in primitive channels or in more complex examples via composition. Indeed, one of our colleagues has demonstrated that it is trivial to embed an Elementary Petri-net into *Reo*, whereas the reverse embedding was much more difficult [30]. Although this embedding not been developed for hybrid Petri nets, or other circuit-based models, its existence indicates that *Reo* can often express more with less.

7 Conclusions and Future Work

Fontana and Buss highlighted the need for an algebraic semantics of behaviour suitable for biological systems [1]. We have proposed that Reo, and the coinductive semantics that underlies it, could provide a suitable framework. We believe this to be the case, because Reo is open-ended, enabling channels with arbitrary behaviour to be added, and it permits the compositional description of biological processes by providing a set of connectors modelling the behaviour present in biological systems. Furthermore, a number of formal tools have been developed to specify and reason about connector behaviour.

The following brief roadmap will guide our future work. We have a number of goals: developing a methodology both for building toolkits for modelling classes of biological systems and for their refinement; extensively studying and validating the approach underlying this methodology *in conjunction with biologists.* It would be great if biologists are willing to work with us to develop more accurate models, so that we can provide better tools. In particular, we would like to further apply the formal reasoning tools for *Reo* with time constraints, so that we can provide models which include metric data such as rates, concentration, delays [22].

Acknowledgements. To Nikolay Daikov, Dennis van der Vlies and Cândida Silva, Jan Rutten and the anonymous referees, who gave detailed comments which significantly helped improve the paper.

References

1. Fontana, W., Buss, L.W.: The barrier of objects: From dynamical systems to bounded organizations. In Casti, J., Karlqvist, A., eds.: Boundaries and Barriers. Addison-Wesley (1996) 56–116
2. Regev, A., Silverman, W., Shapiro, E.: Representation and simulation of biochemical processes using the π-calculus process algebra. In: Pacific Symposium on Biocomputing. Volume 6. (2001) 459–470
3. Danos, V., Krivine, J.: Formal molecular biology done in CCS-R. In: BIO-CONCUR'03. Electronic Notes in Theoretical Computer Science, Marseille, France (2003)
4. Danos, V., Laneve, C.: Formal molecular biology. Theoretical Computer Science **325** (2004) 69–110
5. Danos, V., Pradalier, S.: Projective brane calculus. In: Computational Methods in Systems Biology (CMSB'04). LNCS, Paris, France (2004)
6. Chang, B.Y.E., Sridharan, M.: PML: Towards a high-level formal language for biological systems. In: BIO-CONCUR'03. Electronic Notes in Theoretical Computer Science, Marseille, France (2003)
7. Regev, A., Panina, E.M., Silverman, W., Cardelli, L., Shapiro, E.: BioAmbients: An abstraction for biological compartments. Theoretical Computer Science, Special Issue on Computational Methods in Systems Biology **325** (2004) 141–167
8. Kuttler, C., Niehren, J., Blossey, R.: Gene regulation in the pi calculus: Simulating cooperativity at the lambda switch. In: Workshop on Concurrent Models in Molecular Biology. ENTCS, Elsevier (2004) BIO-CONCUR workshop proceedings.
9. Savageau, M.A.: Rules for the evolution of gene circuitry. In: Pacific Symposium of Biocomputing. (1998) 54–65
10. Kitano, H.: A graphical notation for biochemical networks. BIOSILICO **1** (2003) 169–176
11. Matsuno, H., Doi, A., Nagasaki, M., Miyano, S.: Hybrid Petri net representation of gene regulatory network. In: Proc. Pacific Symposium on Biocomputing 5. (2000) 341–352
12. McAdams, H., Shapiro, L.: Circuit simulation of genetic networks. Science **269** (1995) 650–6
13. Chabrier-Rivier, N., Chiaverini, M., Danos, V., Fages, F., Schächter, V.: Modeling and querying biomolecular interaction networks. Theoretical Computer Science **325** (2004) 25–44
14. Malone, T., Crowston, K.: The interdisciplinary study of coordination. ACM Computing Surveys **26** (1994) 87–119
15. Arbab, F.: Reo: A channel-based coordination model for component composition. Mathematical Structures in Computer Science **14** (2004) 329–366
16. Arbab, F.: Abstract behavior types: A foundation model for components and their composition. In: [31]. (2003) 33–70
17. Arbab, F., Baier, C., Rutten, J.J.M.M., Sirjani, M.: Modeling component connectors in Reo by constraint automata. In: International Workshop on Foundations of Coordination Languages and Software Architectures (FOCSLA). ENTCS, Marseille, France, Elsevier Science (2003)
18. Wolkenhauer, O., Ghosh, B., Cho, K.H.: Control & coordination in biochemical networks (editorial notes). IEEE CSM Special Issue on Systems Biology (2004)
19. Stryer, L.: Biochemistry. Freeman (1988)
20. Department of Energy, U.: `http://www.doegenomestolife.org/` (2004)

21. Ideker, T., Galitski, T., Hood, L.: A new approach to decoding life: systems biology. Annual Review of Genomics and Human Genetics **2** (2001) 343–72
22. Arbab, F., Baier, C., de Boer, F., Rutten, J.: Modeling and temporal logics for timed component connectors. In: IEEE International Conference on Software Engineering and Formal Methods (SEFM '04), Beijing, China (2004) Submitted.
23. Glass, K., Kauffman, S.A.: The logical analysis of continuous, nonlinear biochemical control networks. J. Theoretical Biology **44** (1974) 103–129
24. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
25. Regev, A., Shapiro, E.: Cells as computation. Nature **419** (2002) 343
26. Cardelli, L.: Brane calculi: Interaction of biological membranes. In: Computational Methods in Systems Biology (CMSB'04). Number 3082 in LNCS, Paris, France (2004)
27. Regev, A.: Representation and simulation of molecular pathways in the stochastic pi-calculus. In: 2nd Workshop on Computation of Biochemical Pathways and Genetic Networks. (2001)
28. Schmulevich, I., Dougherty, E.R., Zhang, W.: From boolean to probabilistic boolean networks as models of genetic regulatory networs. Proceedings of the IEEE **90** (2002)
29. Cho, K.H., Wolkenhauer, K.H.J.O.: A hybrid systems framework for cellular processes (2005) BioSystems.
30. Guillen-Scholten, J.V.: A first translation from Reo to Petri nets and vice-versa (2004) Talk at ACG meeting, CWI.
31. de Boer, F., Bonsangue, M., Graf, S., de Roever, W.P., eds.: Formal Methods for Components and Objects. Volume 2852 of LNCS. Springer (2003)

A Rule Markup Language and Its Application to UML

Joost Jacob*

Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands
jacob@cwi.nl

Abstract. In this paper we introduce RML, which stands for Rule Markup Language and is used for rule–based transformations of XML. With RML the user can define *XML wildcard elements*, variables containing parts of the XML such as variables for element names or variables for lists of elements. Any XML vocabulary can be combined with RML to define transformations that can be performed by RML tools also discussed in this paper.

As an application of RML we show how it can be used to specify semantics for statecharts and class–diagrams in UML models. The static structure is defined in XML and the dynamic behavior of the model is captured with RML. The RML tools then provide an XML–based execution platform for UML models. This approach therefore can be used to simulate and analyze UML models.

Keywords: XML, UML, RML, transformation, rules, simulation.

1 Introduction

The work in this paper was initiated and motivated by work in the IST project OMEGA (IST-2001-33522, [OME]) sponsored by the European Commission. The main goal of OMEGA is the correct development of real-time embedded systems in the Unified Modeling Language [UML]. This goal involves the integration of formal methods based on model-checking techniques [Cla] and deductive verification using PVS [PVS].

The eXtensible Markup Language XML (XML [XML]) is used to encode the static structure of UML models in OMEGA. The XML encoding is generated by Computer Aided Software Engineering (CASE) tools; it captures classes, interfaces, associations, state machines, and other software engineering concepts. The OMEGA tools for model-checking and deductive verification are based on a particular implementation of the semantics of the UML models in a tool-specific format ([IF], [Dam], [PVS]). This complicates interoperability of such tools. In order to ensure that these different implementations are consistent, a formal semantics of UML models is developed in OMEGA in the mathematical

* Part of this research was funded by the IST project IST-2001-33522, sponsored by the European Commission.

T. Margaria and B. Steffen (Eds.): ISoLA 2004, LNCS 4313, pp. 26–41, 2006.

formalism of transition systems [PLO]. However, it still requires considerable effort to ensure that these different implementations are indeed compatible with the abstract mathematical semantics. Some of the motivation for RML came in helping with this effort. Since the models produced by the CASE tools are encoded in XML it was a natural choice to look for an XML transformation technique instead of encoding a model and semantics in a special-purpose format. Simulating and analyzing *in* XML adds the interoperability benefit of XML and the many available XML tools can be used on the results.

In this paper a general-purpose method for XML transformations is introduced and its application to the specification and execution of UML models. The underlying idea of this method is to specify XML transformations by means of rules which are formulated in a problem domain XML vocabulary of choice: the rules consist of a mix of XML from the problem domain and the Rule Markup Language (RML, Sect. 3). The input and output of a transformation are pure problem domain XML; RML is only used to help to define transformation rules. The RML approach re-uses the problem domain XML as much as possible, with a "programming by example" technique. With this rule–based approach it becomes possible to define transformations that are very hard to do when using for example XSLT [XSL], the official W3C [W3C] Recommendation for XML transformations, as discussed in Section 2.1.

The RML tools are available as platform-independent command–line tools so they can easily be used together with other tools that have XML as input and output.

RML is not trying to solve *harder* or *bigger* transformations than other approaches. Instead of concentrating on speed or power, RML is designed to be something that is very *usable* and interoperable. Experience in several projects has shown that programmers can learn to use RML in only a few hours with the tutorial that is available online [RML], and even non-programmers put RML to good use. With respect to the RML application to UML models, only knowledge of XML and RML suffices to be able to define and execute their semantics.

As such, RML provides a promising basis for the further development of XML-based debugging and analysis tools for UML models.

XML itself is not intended for human consumption, but we have developed the ASCII Markup Language (AML) representation that helps considerably in this respect. The example model in this paper is presented in AML because AML is more readable than XML, but otherwise equivalent for this purpose. More details about AML and an AML to XML translation, and back, are available at [AML].

Plan of the Paper. The next section starts with describing XML. Section 3 presents RML as a new approach to solve XML transformation problems and describes how to use RML for defining transformation rules. Section 4 shows examples of applications of RML, the main example being an application that results in executable UML models. The conclusion and a discussion of related work is in Sect. 5.

2 XML and XML Transformations

With XML, data can be annotated and structured hierarchically. There are several ways to do this and there is no single best way under all circumstances: designing good XML vocabularies is still an art. For instance, suppose you want to describe a family in XML: a grandmother named Beth, a father named John, a mother name Lucy and son named Bill. One way to do this is:

```
<family>
    <grandma name="Beth" />
    <father name="John" />
    <mother name="Lucy" />
    <son name="Bill" />
</family>
```

The example shows five different XML elements: `family`, `grandma`, `father`, `mother` and `son`. The XML hierarchy is a tree, with nodes called XML elements, and there has to be one and only one XML element that is the root of the tree, in the example the `family` element. An XML element consists of its name, optional attributes and an ordered list of subelements, where a subelement can also be a string. Attributes of XML elements are mappings from keys to values, where the keys are text strings and the values are text strings too.

A string enclosed with angle brackets is called a tag. A minimum tag only contains the element name, like the `<family>` in the example. The element name is not the only thing that can appear between the angle brackets, there can also be attributes like `name="John"` in the example. Attributes consist of the attribute name, an `=` and the attribute value (a text string) enclosed in double quotes.

An XML element that does *not* contain other elements, a so called empty element, has its tag closed by an `/`, as in `<X />`, where `X` is the element name. An XML element that has children consists of two tags: one for the element name (and its attributes), and one for closing the element after its children. In the example the `family` element is the only element with children. There are several rules that define if XML is *well formed*, for instance every opening tag `<X>` has to be closed by a closing tag `</X>`, and these rules can be checked by tools.

But the XML in the example does not reflect the tree like structure of the family. Another way is:

```
<family>
    <female>
        <name>Beth</name>
        <male marriedTo="Lucy">
            <name>John</name>
            <male>
                <name>Bill</name>
            </male>
        </male>
    </female>
    <female marriedTo="John">
        <name>Lucy</name>
    </female>
</family>
```

Here `Beth` is not the value of an attribute but it is the text content of a `name` element. The structure of this example may better indicate that Beth is the mother of John, but the XML is more verbose than the first example.

An XML vocabulary can be formally defined in a DTD (see the XML Specification in [XML]) or an XML Schema [XMS], both W3C Recommendations. There is also an ISO standard for defining vocabularies called RelaxNG [REL]. The definition can express that for instance every `female` in the example must have a `name` child and can have optional `female` or `male` childs. With such a definition, called schema, there are XML tools available that can *validate* if XML is conforming to a schema. Note that validating is different from checking well-formedness. It is possible to refer to the definition of the vocabulary used from inside XML, and there are many more XML concepts that can not be discussed here due to lack of space, for which I refer to the XML Specification [XML].

A schema only defines syntax, the *meaning* of the XML constructs defined has to be defined somewhere else. The W3C is working on standard ways for doing this (viz. the Semantic Web project) but currently this is usually done in plain text documents. The family from the example has a tree like structure so in this respect it is an easy example to describe in XML that also has a tree structure. But a tree is not the only kind of structure that can be modeled conceptually with a schema. Other structures can also be modeled in XML because the XML elements can refer to each other by means of cross-references with identifiers, as in the second example with the `marriedTo` attributes.

A lot of XML vocabularies have been designed in recent years, for all kinds of problem domains. Often these vocabularies are W3C Recommendations, like RDF, XSLT and MathML [Mat]; another example is XMI [XMI] as developed for UML [UML] by the Object Management Group [OMG]. The OMEGA project works with XMI and other XML vocabularies for software.

In general, when having a structure stated in XML data, a dynamics of the structure can be captured by rules for transforming the XML data. The rules that define XML transformations can be stated in XML itself too, but the problem domain XML vocabulary will usually not be rich enough to be able to state a rule. For this it has to be combined with XML that is suitable for expressing transformation rules, containing for example constructs to point out what to replace with what, and where. Section 2.1 shows how an XML vocabulary that is different from the problem domain vocabulary can be used, which is the current way of doing transformations in industry, and Sect. 3 shows the new RML approach that is based on extension.

2.1 XSLT

Extensible Stylesheet Language Transformations (XSLT, [XSL]) is a W3C recommendation for XML transformations. It is designed primarily for the kinds of transformations that have to do with visual presentation of XML data, hence the *style* element in the name. A popular use of XSLT is to transform a dull XHTML page to a colorful and stylized one. Or to generate visualizations from XML data. However, nowadays XSLT is being used more and more for general purpose XML transformations, from XML to XML, but also from XML to text. In the OMEGA project we have used XSLT to do transformations of software models (UML models stated in XMI [XMI]) resulting in models encoded in other XML vocabularies and also resulting in textual syntax like PVS [PVS]. The static structure of the models was transformed. But when we wanted to capture the semantics of execution of these software models we found that XSLT is not very usable for the particular kind of transformations that describe dynamics. These transformations use a match pattern that is distributed over several parts of an XML tree, whereas the matching technique used in XSLT is designed to match in a linear way, from root to target node in a tree. This linear matching is not suitable for matching of a pattern with several branches.

For instance, matching duplicate children of an element is very hard with XSLT. The MathML expression

```
<math>
    <apply>
        <and />
        <ci>p</ci>
        <ci>p</ci>
        <ci>q</ci>
    </apply>
</math>
```

meaning $p \wedge p \wedge q$ in propositional logic, is logically equivalent to

```
<math>
    <apply>
        <and />
        <ci>p</ci>
        <ci>q</ci>
    </apply>
</math>
```

meaning $p \wedge q$. In the MathML the `<ci>` element is used for pointing out constant identifiers and the `apply` element is used for building up mathematical expressions. Suppose we would like to transform all $p \wedge p \wedge q$ into $p \wedge q$, where p and q can be anything but two p's in an expression are equal. To perform such a transformation a tool has to look for a pattern with two identical children and then remove one of the children. Since XSLT is a Turing–complete functional programming language, it *is* possible to do this transformation, but XSLT templates for these kinds of transformations are extremely long and complex. XSLT simply was not designed for these kinds of transformations; the designers did not feel much need for them in the webwide world of HTML and webpublishing. The MathML+RML rule for removing duplicate children is simple, it is one of the examples in the RML tutorial [RML].

3 RML

This section first introduces the idea of *XML wildcard elements*. After that the RML syntax is introduced in Sect. 3.2, before Section 3.3 describes the RML tools.

3.1 XML Wildcard Elements

The transformation problem as shown in Sect. 2.1 reminds one of the use of wildcards for solving similar problems in text string matching. The idea of using an XML version of wildcards is a core idea of our method and of this paper. The idea is to define XML notation for an XML version of constructs like the `*` and `?` and `+` and others in text–based wildcards as in Perl regular expressions, and then using these constructs for matching and variable binding. These constructs are called *XML wildcard elements*. They consist of complete XML elements and attributes as can be formally defined with a schema, but they also consist of *extensions* for denoting wildcard variables *inside* a problem domain XML. These variables are just like the variables as they are used in the various languages available for text–based wildcard matching, for instance Perl regular expressions. The variables have a name, and they are given a value when a match succeeds. This value can then be used in the output of a transformation rule.

3.2 The RML Syntax

RML rules are stated in XML. The basis of a rule is that in the antecedent of the rule the input is matched, and then whatever matched is replaced by the consequent of the rule.

RML was designed to be *mixed* with any problem domain XML, to be able to define transformations while re-using the problem domain XML as much as possible. RML is a mixture of XML elements, conventions for XML-attribute names, and conventions for attribute values, to mix in with XML from the problem domain vocabulary at hand. RML introduces some new XML elements and uses an element from XHTML. From XHTML only the `div` element is used and it is used to distinguish a rule, the antecedent of a rule, and the consequence of a rule by means of the `class` attribute of the `div` tag. We use the `div` tag from XHTML for reasons that have to do with a presentation in browsers.

The Table in Fig. 1 lists all the current RML constructs with a short explanation of their usage in the last column. These have been found to be sufficient for all transformations encountered so far in practice in the projects where RML is being used. An `X` in the XML tags can be replaced by a string of choice. The *position* that is sometimes mentioned in the explanations is the position in the sequential list that results from a root-left-right tree traversal of the XML tree for the rule. It corresponds with how people in the western world reading an XML document encounter elements: top-down and left to right. A position in the rule tree corresponds with zero or more positions in the input tree, just like the `*` in the wildcard expression `a*b` corresponds with `c` on input `acb` and with `cd` on input `acdb` and with nothing on input `ab`. With the constructs in the Table in Fig. 1 the user can define variables for element names, attribute names, attribute values, whole elements (with their children) and lists of elements. An XML+RML version of the `a*b` wildcard pattern is

```
<a /> <rml-list name="Star" /> <b />
```

Elements that designate rules	
div	class="rule"
div	class="antecedent" context="yes"
div	class="consequence"

element	attribute	A	C	meaning
Elements that match elements or lists of elements				
rml-tree	name="X"	*		Bind 1 element at this position to RML variable X.
rml-text	name="X"	*		Bind XML text-content to variable X.
rml-list	name="X"	*		Bind a sequence of elements to X.
rml-use	name="X"		*	Output the contents of the RML variable X at this position.
Matching element names or attribute values				
rml-X	...	*		Bind element name to RML variable X.
rml-X	...		*	Use variable X as element name.
...	...="rml-X"	*		Bind attribute value to X.
...	...="rml-X"		*	Use X as attribute value.
...	rml-others="X"	*		Bind *all* attributes that are not already bound to X.
...	rml-others="X"		*	Use X to output attributes.
...	rml-type="or"	*		If this element does not match, try the next element in the antecedent if that also has rml-type="or".
Elements that add constraints				
rml-if	child="X"	*		Match if X is already bound to 1 element, and occurs *somewhere* in the current sequence of elements.
rml-if	nochild="X"	*		Match if X does not occur in the current sequence.
rml-if	last="true"	*		Match if the preceding sibling of this element is the last in the current sequence.

A * in the A column means the construct can appear in a rule antecedent. A * in the C column is for the consequence.

Fig. 1. All the RML constructs

and this can be used as part of the antecedent of an RML rule that uses the contents of the Star variable in the consequent of the rule. The a and b elements are from some XML vocabulary, the rml-list element is from RML and described in the Table in Fig. 1. Section 4.1 shows a small but complete RML rule. Examples of input and rules take up much space and although we would have preferred to present more rule examples here now, there is simply not enough space to do that and we strongly invite the reader to look at the examples in the RML tutorial at [RML] where there are examples for element name renaming, element replacing, removing duplicates, copying, attribute copying, adding hierarchy and many more.

It is easy to think of more useful elements for RML than in the Table, but not everything imaginable is implemented because a design goal of RML is to keep it as simple and elegant as possible. Only constructs that have proven themselves useful in practice are added in the current version.

The execution of a rule consists of binding variables in the matching process, and then using these variables to produce the output. Variable binding in RML happens in the order of a root-left-right traversal of the input XML tree. If an input XML tree contains more than one match for a variable then only the first match is used for a transformation. The part of the input that matches the rule antecedent is *replaced* by the consequent of the rule. If a rule does not match then the unchanged input is returned as output. If a rule matches input in more than one place and you want to transform all matches then you will have to repeat applying the rule on the input until the output is stable. There is a special RML tool called `dorules` for this purpose.

3.3 The RML Tools and Libraries

Open-source tools and libraries can be downloaded from the RML website [RML] where also an RML tutorial can be found. The tutorial contains information about installing and running the tool, and there is also more technical information on the website, for instance about a matching algorithm. The tools and libraries have been successfully used under Windows, Linux, Solaris, and Apple.

A typical usage pattern of the `applyrule` command–line tool is

```
$ python applyrule.py --rule myrule.xml --input myinput.xml
```

that will print the result to console, or it can be redirected to a file.

The rule and the input are parameterized, not only as command–line parameters but also in the internal `applyrule` function; this makes the tool program also usable as a library and thus suitable for example for programming a simulation engine. There is an interface to additional hook functions so tools can be extended, for instance for programming new kinds of constraints on the matching. However, the set of RML constructs in the current version has proven to be sufficient for various XML transformation work, so adding functions via the hooks will normally not be necessary. An example of when it *is* desirable to add functions is when a tool designer for example wants to add functionality that does calculations on floating point values in the XML. The RML tools are written in Python [Pyt] and the Python runtime can use the fast `rxp` parser written in, and compiled from, C, so the XML parsing, often the performance bottleneck in such tools, is as fast and efficient as anything in the industry. If the rxp module is not installed with your Python version then RML automatically uses another XML parser on your system.

The RML tutorial at [RML] also describes a very simple XML vocabulary for defining RML recipes, called Recipe RML (RRML), and a tool called `dorecipe` for executing recipe–based transformations. RRML is used to define sequences of transformations and has proven itself useful in alleviating the need for writing shell scripts, also called batch files, containing sequences of calls to the RML tools.

4 RML Examples

The main example presented in this paper is the executable specification of the semantics of UML models in XML and this is the topic of Section 4.1. Section 4.2 briefly mentions other projects wherein RML is applied.

4.1 Executable UML Models

The application of RML to the semantics of UML models and its resulting execution platform is based on the separation of concerns betweeen coordination/communication and computation. This exploits the distinction in UML between so-called triggered and primitive operations. The behavior of classes is specified in UML statemachines with states and transitions, and every transition can have a trigger, guard, and action. A transition does not need to have all three, it may for example have only an action or no trigger or no guard. Triggered operations are associated with events: if an object receives an event that is a trigger for a transition, and the object is in the right location for the transition, and the guard for that transition evaluates to True, then the action that is specified in the transition is executed. The triggered operations can be synchronous (the caller blocks until an answer is returned) or asynchronous. Events can be stored in event queues, and the queues can be implemented in several ways (FIFO, LIFO, random choice, ...). There are also primitive operations: they correspond to statements in a programming language, without event association or interaction with an event mechanism. The primitive operations are concerned with computations, i.e. data-transformations, the triggered operations instead are primarily used for coordination and communication. More details can be found in [Dam].

This distinction between triggered and primitive operations and the corresponding separation of concerns between coordination/communicaion and computation is reflected in the RML specification and execution of UML models which delegates (or defers) the specification of the semantics and the execution of primitive operations to the underlying programming language of choice. This delegation is not trivial, because the result of primitive operations has to be reflected in the values of the object attributes in the XML, but the details of the delegation mechanism can not be given here due to a lack of space.

In our example the problem domain is UML and we will use a new XML vocabulary that is designed for readability and elegance. This language is called *km*, for kernel model; a RelaxNG [REL] schema is at http://homepages.cwi.nl/˜jacob/km/km.rnc.

The online example is a prime sieve, it was chosen because it shows all the different kinds of transitions and it has dynamic object creation. It generates objects of class `Sieve` with an attribute `p` that will contain a prime number. But the user can edit the example online or replace it with his or her own example, if the implementation language for actions and guards is the Python programming language. A similar application can be written for the Java language, and UML models from CASE tools can be translated automatically to the km language.

The example can be executed online in an interactive webapplication on the internet at [KM]. In the km application the user fills in a form with an object identity and a transition identity, and pressing a button sends the form to the webapplication that performs the corresponding transition. Instead of a user filling in a form, a program can be written that calls the website and fills in the form, thus automating the tool. We did so, but for this paper we consider a discussion of the automated version out of scope.

The km language defines XML for class diagrams and object diagrams. The classes consist of attribute names and a statemachine definition. The statemachines have states and transitions, where the transitions have a guard, trigger and action like usual in UML. The objects in the object diagram have attributes with values and an event queue that will store events sent to the object. An example of an object is

```
objectdiagram
    obj class=Sieve id=2 location=start target=None
        attr name=p value=None
        attr name=z value=None
        attr name=itsSieve value=None
        queue
            op name=e
                param value=2
```

where the object is of type `Sieve`, finds itself in the `start` state of the statemachine of the `Sieve` class, and has an eventqueue with one event in it with name `e` and event parameter `2`.

A detailed description of the km language and its design would take too much space here, but the interested reader who knows UML will have no trouble recognizing the UML constructs in the models since the km language was designed for readability.

In the km language the event semantics is modelled, but the so-called *primitive* operations that change attribute values are deferred to a programming language. So the models will have event queues associated with objects and executing a model will for example show events being added to queues, but operations that are not involved with events but only perform calculations are stored in the model as strings from the programming language of choice. Such an operation can be seen in the example as

```
transition id=t3
    source state=state_3
    target state=state_1
    action
        implementation
            """x = x + 1"""
```

where we see a transition in the statemachine with an action, the statement executed by the programming language (Python in this case) is `x = x + 1`. Transitions can also have a guard with an expression in a programming language, also encoded as text content of an `implementation` element.

We can now show a simple example RML rule.

```
<div class="rule" name="set location">
  <div class="antecedent">
    <obj id="rml-IDOBJ" location="rml-L" target="rml-T" rml-others="rml-O" >
      <rml-list name="ObjChildren"/>
    </obj>
  </div>
  <div class="consequence">
    <obj id="rml-IDOBJ" location="rml-T" target="None" rml-others="rml-O">
      <rml-use name="ObjChildren"/>
    </obj>
  </div>
</div>
```

This is a rule that is used after a transition has been taken successfully by an object modeled with km. With this rule the `location` attribute of the object is assigned the value of the `target` attribute and the `target` attribute is set to `None`. An example of the effect of the rule would be that

```
<obj id="id538" location="state_3" target="state_5" ... >
  <queue>
    ...
  </queue>
</obj>
```

is changed into

```
<obj id="id538" location="state_5" target="None" ... >
  <queue>
    ...
  </queue>
</obj>
```

for an object with identifier `id538`.

When applying this rule, the RML transformation tool first searches for an `obj` element in the input, corresponding with the `obj` element in the `antecedent` of the rule. These `obj` elements match if the `obj` in the input has an `id` attribute with the value bound to the RML `IDOBJ` variable mentioned in the antecedent, in the example this value is `id538` and it is bound to the RML variable `IDOBJ` *before* the rule is applied. This pre-binding of some of the variables is how the tool can manage and schedule the execution of the RML transformation rules. The `IDOBJ` is a value the user of the online webapplication supplies in the form there. If the `obj` elements match, then the other RML variables (`L`, `T`, `O` and `ObjChildren`) are filled with variables from the input `obj`. The `L`, `T` and `O` variables are bound to strings, the `ObjChildren` variable is bound to the children of the `obj` element: a list of elements and all their children. The `consequence` of the rule creates a new `obj` element, using the values bound to the RML variables, and *replaces* the `obj` element in the input with this new `obj` element.

Due to lack of space we restrict the description of the formalization in RML to the rule for the removal of an event from the event-queue, the antecedent is shown in AML notation:

```
km
    classdiagram
        ...
        class name=rml-ClassName
            statemachine
                transition id=rml-IDTRANS
                    trigger
                        op name=rml-TriggerName
                            rml-list name=Params
                ...
    objectdiagram
        ...
        obj class=rml-ClassName id=rml-IDOBJ
                        rml-others=rml-OtherObjAttrs
            queue
                rml-list name=PreEvents
                op name=rml-TriggerName
                rml-list name=PostEvents
```

and this contains some lines with ... in places where rml-list and rml-use constructs are used to preserve input context in the output. Here we see that in RML a pattern can be matched that is distributed over remote parts in the XML, the remoteness of the parts is why the rule has so many lines. In short, this rule looks for the name of the trigger that indicates the event that has to be removed from the event-queue, and then simply copies the event-queue without that event. But to find that name of the trigger, a search through the whole km XML model has to take place, involving the following steps.

During application of this rule, the matching algorithm first tries to match the input with the antecedent of the rule, where IDOBJ and IDTRANS are pre-bound RML variables, input to the tool. With these pre-bound variables it can find the correct obj, then it finds the ClassName for that object. With the ClassName the class of the object can be found in the classdiagram in km XML. When the class of the object is found, the transition in that class with id TRANSID can be found and in that transition element in the input we can finally find the desired TriggerName. The algorithm then looks for an op (operation) event with name TriggerName in the event-queue of the obj, and binds all other events in the event-queue to RML variables PreEvents and PostEvents. In the consequence of the rule then, all these bound RML variables are available to produce a copy of the input, with the exception that the correct event is removed. As given, the rule removes the first event that matches. It is trivial to change the rule to one that removes only the first event in a queue (by removing the PreEvents), or only the last. This is an example that shows that the semantics defined in the RML rules can be easily adapted, even *during* a simulation, and this makes such rules particulary suitable for experimental analysis.

The km application gives comments, for example about the result of the evaluation of a guard of a transition. If the user for instance selects a transition identity that does not correspond with the current state of the object, in the online example if you select (ObjID,TransitionID)=(1,t1) twice, a message is displayed on top of the model
`# Exiting: Wrong location (object:state_1 transition:start)` meaning that transition t1 can not be taken because the object is in state state_1 and the transition is

defined for a `source` state with name `start`. Such messages do not interfere with the model itself, they are encoded as comments, and the model is unchanged after this message.

The only software a user needs to use the interactive application is a standards compliant browser like Mozilla or Internet Explorer. A user can not only go forward executing a model, but also go *backward* with browser's Back button. This is an example of the benefit of interoperability that XML offers, together with a software architecture and design that is platform independent.

4.2 Other Examples

If a formalism is expressed in mathematics, then MathML is a generally usable way to express structure, and RML rules extending MathML can capture the dynamics. As an example of this, RML is used for an online interactive theorem prover that can be used to derive proofs for tautologies in propositional logic, at http://homepages.cwi.nl/˜jacob/MathMLcalc/MathMLcalc.html.

Although defining models and semantics in MathML will appeal to the mathematically educated, sometimes it is better to define a new special-purpose XML vocabulary; to make it more concise, better readable, more efficient, and for several other reasons. This was the case in the Archimate [Arch] project where RML has been applied successfully to Enterprise Architecture and Business Models. Rule–based transformations are being used for analysis of models and for visualizations. The RML tutorial and the downloadable RML package at [RML] contain examples in the Archimate language.

5 Related Work and Conclusion

Standards related to RML are XML [XML], MathML [Mat] and XSLT [XSL]. MathML is a W3C specification for describing mathematics in XML, and it is the problem domain language for the proof example in this paper. XSLT is a W3C language for transforming XML documents into other XML documents, and is discussed in Section 2.1. There are also standards that are indirectly connected with XML transformations, like XQuery that can treat XML as a database that can be queried, but a discussion of the many XML standards here is out of scope.

The RuleML [RUL] community is working on a standard for rule-based XML transformations. Their approach differs from the RML approach: RML re-uses the problem domain XML, extended with only a few constructs (in the table in Fig. 1) to define rules; whereas RuleML superimposes a special XML vocabulary for rules. This makes the RuleML approach complex and thus difficult to use in certain cases. The idea of using wildcard elements for XML has not been incorporated as such in the RuleML approach, but perhaps it can be added to RuleML and working together with the RuleML community in the future can be interesting.

There are a number of tools, many of them commercial, that can parse XML and store data in tables like those in a relational database. The user has to define rules for extracting the data, to define what is in the columns and the rows of the tables, to define an entity-relationship model, and other things. Once the data the user is interested in is in the database, a standard query language like SQL can be used to extract data. And then that data can be used to construct new XML. The XML application called XQuery can be used in a similar way, and it is the approach taken by ATL [Bez]. It would be possible to do any transformation with these techniques, but it would be very complex.

The Relational Meta-Language [RelML] is a language that is also called RML, but intended for compiler generation, which is much more roundabout and certainly not usable for rapid application development like with RML in this paper.

An example of another recent approach is fxt [Ber], which, like RML, defines an XML syntax for transformation rules. Important drawbacks of fxt are that it is rather limited in its default possibilities and relies on hooks to the SML programming language for more elaborate transformations. For using SML a user has to be proficient in using a functional programming language. An important disadvantage of a language like SML is that it is not a mainstream programming language like Python with hundreds of thousands or users worldwide, which makes it unattractive to invest in tools based on SML. The fxt tools are available online but installing them turned out to be problematic.

The experience with several tools as mentioned above leads to the concept of *usability* of a tool in general. Here, a tool is not considered usable enough if it is too difficult to install and configure it and get it to run, or if the most widely used operating system Windows is not supported, or if working with the tool requires a too steep or too high learning curve, for example because the user has to learn a whole new programming language that is not a mainstream programming language. Although the fxt article [Ber] interestingly mentions "XML transformation ... for non-programmers", fxt is unfortunately an example of an approach that is not usable enough according to this usability definition.

XML is still gaining momentum and becoming more important and as a result there are many more tools from academic research available, rather too much to mention here as an internet search for “XML tool” reveals hundreds of search results. Unfortunately none of them turned out to be useful in practice for our work according to the above definition of usability, after spending considerable time trying them out.

Other popular academic research topics that could potentially be useful for rule–based XML transformations are term–rewriting systems and systems based on graph grammars for graph reduction. However, the tested available tools for these systems suffer from the same kind of problems as mentioned above: the tools are generally not portable and most will never be portable for technical reasons, and using these tools for XML transformations is an overly complex way of doing things. To use these kind of systems, there has to be first a translation from the problem XML to the special-purpose data structure of the system. And only then, in the tool–specific format, the semantics is defined. But the techniques used in these systems are interesting, especially for very complex or hard transformations, and it looks worthwhile to see how essential concepts of these techniques can be incorporated in RML in the future.

Compared with the related work mentioned above, a distinguishing feature of the RML approach is that RML *re-uses* the language of the problem itself for matching patterns and generating output. This leads in a natural way to a much more usable and clearly defined set of rule–based transformation definitions, and an accompanying set of tools that is being used successfully in practice.

References

[AML] *AML website*. url: http://homepages.cwi.nl/˜jacob/aml

[Arch] *The Archimate project*. url: http://www.telin.nl/NetworkedBusiness/Archimate/ENindex.htm

[Ber] A. Berlea and H. Seidl. *fxt A transformation language for XML documents*. Journal of Computing and Information Technology, 10(1):19–35, 2002.

[Bez] J. Bzivin, G. Dup, F. Jouault, G. Pitette, and J. Rougui. *First experiments with the ATL model transformation language: Transforming XSLT into XQuery*. OOPSLA 2003 Workshop, Anaheim, California, October 27, 2003. url: http://modelware.inria.fr/rubrique12.html

[Cla] E. M. Clarke, E. A. Emerson, and A. P. Sistla. *Automatic verification of finite-state concurrent systems using temporal logic specifications*. In ACM Transactions on Programming Languages and Systems, 8(2):244–263, 1986.

[Dam] W. Damm, B. Josko, A. Pnueli, and A. Votintseva. *Understanding UML: a formal semantics of concurrency and communication in real-time UML*. In Proceedings of Formal Methods for Components and Objects (FMCO), LNCS 2852, 2002.

[IF] M. Bozga, S. Graf, and L. Mounier. *Automated validation of distributed software using the IF environment*. In S. D. Stoller and W. Visser, editors, Workshop on Software Model-Checking, associated with CAV'01 (Paris, France) July 2001 Volume 55 of Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers.

[KM] *UML Kernel model semantics demonstration*. url: http://homepages.cwi.nl/˜jacob/km/cgikm.html

[Mat] *Mathematical markup language (MathML) Version 2.0 (2nd Edition)*, W3C recommendation, August 2003. url: http://www.w3.org/Math/

[OME] *OMEGA project: correct development of real-time embedded systems*. url: http://www-omega.imag.fr

[OMG] *The object management group (OMG)*. url: http://www.omg.org/

[PLO] G.D. Plotkin. *A structural approach to operational semantics*. Technical Report DAIMI FN 19, Department of Computer Science, University of Aarhus, Denmark, 1981.

[PVS] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS system guide version 2.4*, December 2001. url: http://pvs.csl.sri.com/

[Pyt] G. van Rossum. *Python reference manual*. Centrum voor Wiskunde en Informatica (CWI), report ISSN 0169-118X, April, 1995. url: http://www.python.org/

[REL] J. Clark. *The design of RELAX NG*. December 6, 2001. url: http://www.thaiopensource.com/relaxng/design.html

[RelML] M. Pettersson. RML - A new language and implementation for natural semantics. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming, PLILP, volume 884 of LNCS*, pages 117-131. Springer-Verlag, 1994.

[RML] *RML website*. url: http://homepages.cwi.nl/˜jacob/rml

[RUL] *The rule markup initiative community*. url: http://www.dfki.uni-kl.de/ruleml/

[UML] G. Booch, I. Jacobson, and J. Rumbaugh. *The unified modeling language reference manual*. Addison Wesley, 1999.

[W3C] *World wide web consortium (W3C)*. url: http://www.w3c.org

[XMI] *XML metadata interchange (XMI) v2.0*, OMG, May 2003. url: http://www.omg.org/technology/documents/formal/xmi.htm

[XML] *Extensible markup language (XML) 1.0 (Second Edition)*, W3C recommendation, October 2000. url: http://www.w3c.org/XML/

[XMS] R. L. Costello. *XML schema tutorial*, W3C, September 2001. url: http://www.w3.org/XML/Schema

[XSL] *XSL transformations (XSLT) version 1.0*, W3C recommendation, November 1999. url: http://www.w3c.org/TR/xslt

Using XML Transformations for Enterprise Architectures

A. Stam[1,⋆], J. Jacob[2], F.S. de Boer[1,2],
M.M. Bonsangue[1,⋆⋆], and L. van der Torre[3]

[1] LIACS, Leiden University, The Netherlands
astam@liacs.nl
[2] CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
[3] University of Luxembourg, Luxembourg

Abstract. In this paper we report on the use of XML transformations in the context of Enterprise Architectures. We show that XML transformation techniques can be applied to visualize and analyze enterprise architectures in a formal way. We propose transformational techniques to extract views from an XML document containing architectural information and indicate how to perform a specific form of impact analysis on this information. The transformations are formally expressed with the language RML, a compact yet powerful transformation language developed at CWI, which obtains its power from regular expressions defined on XML documents. We discuss a tool that has been built on top of it to visualize the results of the transformations and illustrate the advantages of our approach: the genericity of XML, the application of a single technique (namely XML transformations) for various tasks, and the benefits of having a model viewer which is in complete ignorance of the architectural language used.

1 Introduction

In this paper, we investigate the use of XML transformation techniques in the context of enterprise architectures, a field in which much work is currently often done by hand, like analysis and the time-consuming creation of views for different stakeholders. The absence of formal methods and techniques hinders the quality of analyses, the consistency between different views and the agility with respect to changes in the architecture. Moreover, architects often want to have their own style of visualization (for cultural and communication reasons within organizations), without having to conform to a certain standard. Many architectural tools depend on a specific architectural language and often only support one visualization style or visual language for it.

In order to overcome these problems, we propose a way of working that introduces flexibility in the architectural language and visualizations used, while

[⋆] Corresponding author.

[⋆⋆] The research of Dr. Bonsangue has been made possible by a fellowship of the Royal Netherlands Academy of Arts and Sciences.

T. Margaria and B. Steffen (Eds.): ISoLA 2004, LNCS 4313, pp. 42–56, 2006.

adding formality to both the actual creation of views and the analysis of architectures. Our approach is to specify architectural information as XML documents and use XML transformation techniques for creating views and performing analyses on an architecture. Moreover, we propose to use a visualization tool which is independent of a specific architectural or visual language.

In particular, within this paper, we focus on three subquestions:

- Given a set of architectural information described in a single XML document. How can we use XML transformations to select a subset of this information for a specific architectural view?
- How can we transform an XML document containing architectural information into another XML document containing visual information in terms of boxes, lines, etc.? How can we build a *model viewer* which interprets this visual information without having to know anything about the architectural language used?
- How can we use XML transformations to perform analyses on an architectural description? We have chosen to focus at a specific form of *impact analysis*: given an entity within the architectural description which is considered to be modified or changed, which other entities in the description are possibly influenced by this change?

We have carried out the following activities: First, we developed a running example for verification of our ideas and techniques. Then, we developed an XML document containing the architectural information of the running example. We used the ArchiMate language and its corresponding XML Schema, containing the concepts from the ArchiMate metamodel. After this, we developed an XML Schema for visualization information and built a *model viewer*, which is entirely ignorant of the ArchiMate language, and only interprets visualization information and shows this information on the screen. We only used XML transformation techniques for the actual visualization. Then, we selected an easy-to-use transformation tool, namely the *Rule Markup Language* (RML), and built the transformation rules for selection, visualization and impact analysis.

The layout of this document is as follows: In Section 2, we introduce the reader to enterprise architecture and ArchiMate. In Section 3, XML and the Rule Markup Language (RML) are explained. In Section 4 we introduce the running example: ArchiSurance, a small insurance company which has the intention to phase out one of its core applications. In Section 5 we show transformation rules for the creation (selection and visualization) of architectural views, while in Section 6 we illustrate transformation techniques for analysis by means of performing a small impact analysis. In Section 7 we conclude.

2 Enterprise Architecture

A definition of architecture quoted many times is the following IEEE definition: "the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution" [11]. Therefore, we can define enterprise architecture [8] as the fundamental organization of an enterprise embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution. It covers principles, methods and models for the design and implementation of business processes, information systems, technical infrastructure and organizational structure.

Architectural information is usually presented via (architectural) views. With these views and the information they contain, stakeholders within an organization are able to gain insight into the working of the organization, the impact of certain changes to it, and ways to improve its functioning. Usually, we can distinguish between architectural information in relation to the as-is situation of an organization and information in relation to its intended to-be situation. According to IEEE, views conform to viewpoints that address concerns of stakeholders.

2.1 ArchiMate

Within the ArchiMate project [10], a language for enterprise architecture has been developed [7]. This language can be used to model all architectural aspects of an organization. A metamodel containing the concepts in the ArchiMate language is given in Figure 1.

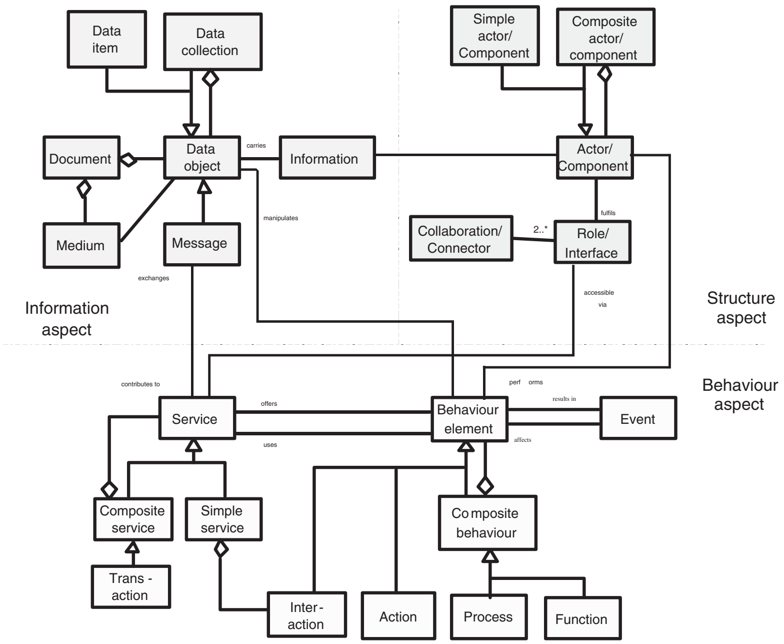


Fig. 1. The ArchiMate metamodel

As can be seen, the language contains concepts for several aspects of an organization. The individual concepts can be specialized for multiple domains, like the business domain, application domain or technical domain. Thus, a *Service* can be a business service, an application service or a technical service, for example.

3 XML and the Rule Markup Language

In this section we introduce XML and present the Rule Markup Language (RML), which is the XML transformation language we have used for creating views and performing analyses of enterprise architectures.

3.1 XML

The eXtensible Markup Language (XML) [4] is a universal format for documents containing structured information so that they can be used over the Internet for web site content and several kinds of web services. It allows developers to easily describe and deliver rich, structured data from any application in a standard, consistent way.

Today, XML can be considered as the lingua franca in computer industry, increasing interoperability and extensibility of several applications. Terseness and human-understandability of XML documents is of minimal importance, since XML documents are mostly created by applications for importing or exporting data.

3.2 The Rule Markup Language

In our study, we have used the Rule Markup Language (RML) for the specification of XML transformations. From a technical point of view, other transformation tools like the popular XSLT, RuleML or QVT could have been used as well for the applications described in this paper. However, RML was originally designed to make the definition of executable XML transformations possible for various stakeholders other than programmers, including architects. Thus, instead of creating views and performing analyses by hand, architects can formally specify transformations of the contents of their own XML documents and apply these transformations in order to select, visualize and analyze architectural information.

The Rule Markup Language (RML) [6] consists of a set of XML constructs which can be added to an existing XML vocabulary in order to define RML rules for that XML vocabulary. Specific RML tools can execute these rules, to transform the input XML according to the rule definition. The set of RML constructs is shown in Table 2 with a short explanation of each construct.

Each RML rule consists of an antecedent and a consequence. The antecedent defines a pattern and variables in the pattern. Without the RML constructs for variables, this pattern would consist only of elements from the chosen XML vocabulary. The pattern in the antecedent is matched against the input XML.

Elements that designate rules	
`div`	`class="rule"`
`div`	`class="antecedent" context="yes"`
`div`	`class="consequence"`

element	attribute	A	C	meaning
Elements that match elements or lists of elements				
`rml-tree`	`name="X"`	*		Bind 1 element (and children) at this position to RML variable X.
`rml-list`	`name="X"`	*		Bind a sequence of elements (and their children) to X.
`rml-use`	`name="X"`		*	Output the contents of RML variable X at this position.
Matching element names or attribute values				
`rml-X`	...	*		Bind element name to RML variable X.
`rml-X`	...		*	Use variable X as element name.
...	`...="rml-X"`	*		Bind attribute value to X.
...	`...="rml-X"`		*	Use X as attribute value.
...	`rml-others="X"`	*		Bind *all* attributes that are not already bound to X.
...	`rml-others="X"`		*	Use X to output attributes.
...	`rml-type="or"`	*		If this element does not match, try the next one with rml-type=”or”.
Elements that add constraints				
`rml-if`	`child="X"`	*		Match if X is already bound to 1 element, and occurs *somewhere* in the current sequence of elements.
`rml-if`	`nochild="X"`	*		Match if X does not occur in the current sequence.
`rml-if`	`last="true"`	*		Match if the younger sibling of this element is the last in the current sequence.
A `*` in the `A` column means the construct can appear in a rule antecedent. A `*` in the `C` column is for the consequence.				

Fig. 2. An overview of the RML constructs

RML constructs can contain variables, which are specified in a familiar way, by using wild-card patterns like `*` and `+` and `?`. The RML variables also have a *name* that is used to remember the matching input. Things that can be stored in RML variables are element names, element attributes, whole elements (including the children), and lists of elements.

If the matching of the pattern in the antecedent succeeds, the variables are bound to parts of the input XML and they can be used in the consequence of an RML rule to produce output XML. When a rule is applied to the input, one of the RML tools will by default replace the part of the input that matched the antecedent, by the output defined in the consequence of the rule; the input surrounding the matched part remains unchanged.

RML does not define, need, or use another language, it only adds a few constructs to the XML vocabulary used, like the wild-card pattern matching.

3.3 Comparison with Other Techniques

XSLT[5] is a W3C language for transforming XML documents into other XML documents.

The *RuleML* [3] community is working on a standard for rule-based XML transformations. Their approach differs from the RML approach: RuleML superimposes a special XML vocabulary for rules, which makes it less suitable to use in combination with another XML vocabulary.

The *Relational Meta-Language* [9] is a language that is also called RML, but intended for compiler generation, which is definitely not suited for rapid application development like with RML in this paper.

Another recent approach is *fxt* [1], which, like RML, defines an XML syntax for transformation rules. Important drawbacks of fxt are that it is rather limited in its default possibilities and relies on hooks to the SML programming language for more elaborate transformations.

Other popular academic research topics that could potentially be useful for rule based XML transformations are term rewriting systems and systems based on graph grammars for graph reduction. However, using these tools for XML transformations is a contrived and arbitrary way of doing things. To exploit these kind of systems successfully in a context where a specific XML vocabulary is already in use, there has to be first a translation from this specific XML to the special-purpose data structure of the system. And only then, in the tool–specific format, the semantics is defined. But the techniques used in these systems are interesting, especially for very complex transformations.

Compared with the related work mentioned above, a distinguishing feature of the RML approach is that RML *re-uses* the language of the problem itself for matching patterns and generating output. This leads in a natural way to a much more usable and clearly defined set of rule based transformation definitions, and an accompanying set of tools that is being used successfully in practice.

4 Running Example

Throughout this paper, we use a running example to illustrate our ideas. Though our example is small compared to a real-life enterprise architecture, its sole purpose in this paper is to illustrate the use of XML transformation techniques, for which it contains sufficient complexity. Analyzing the performance of RML when applied to larger architectural models is a topic for future research.

A small company, named ArchiSurance, sells insurance products to customers. Figure 3 contains a Business View of the company. Two roles are involved, namely the insurance company and the customer, which work together in two collaborations, namely negotiation, which is the set of activities performed in order to come to an appropriate insurance for a customer by discussion and consultation, and contracting, i.e., the set of activities performed in order to register a new customer and let it sign a contract for an insurance policy.

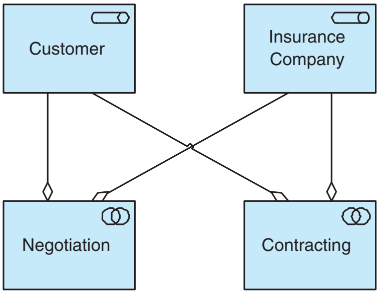


Fig. 3. A Business View of ArchiSurance

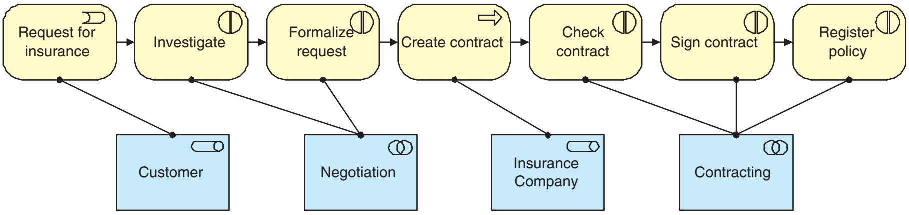


Fig. 4. A Process View of ArchiSurance

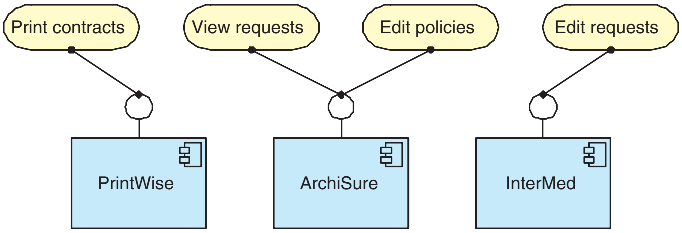


Fig. 5. An Application View of ArchiSurance

Within Figure 4, the business process for selling an insurance product to a customer is shown in a Process View, together with the roles and collaborations that are involved in executing the individual steps within the process.

Figure 5, an Application View, shows the software products (components) that are used within the ArchiSurance company and the services they offer. *ArchiSure* is a custom-made software application for the administration of insurance products, customers and premium collecting. *PrintWise* is a out-of-the-box tool for official document layout and printing. *InterMed* is an old application, originally meant for intermediaries to have the possibility to enter formal requests for insurance products for their customers. The application is now used by employees of the insurance company, since no intermediaries are involved in

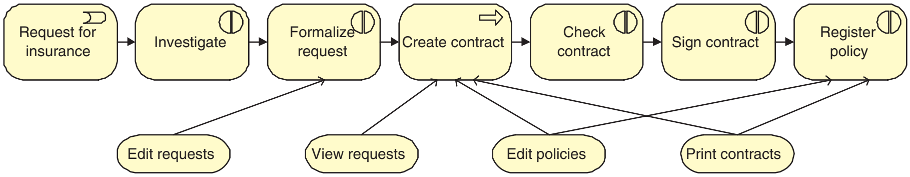


Fig. 6. A Service View of ArchiSurance

selling insurance products anymore. Actually, the company would like to phase out this application.

In Figure 6, a Service View is presented: the process for selling products is shown again, now together with the services that are used within each step.

4.1 An XML Description of the Example

Although the four views (Business View, Process View, Application View, Service View) are depicted separately, they are clearly related to each other via the concepts they contain. In this small example, it is possible to imagine the *big picture* in which all ArchiSurance information is contained.

Within the ArchiMate project, an XML Schema has been developed which can be used for storage or exchange of architectural information. Based on this Schema, we have created a single XML document that contains all information about ArchiSurance. The use of a single document is no problem in this small case, but when architectural models are larger, a single document could be difficult to maintain. Investigating the use of more XML documents for a single architectural model is a topic for future research.

For illustration, a fragment of the XML document is shown below. It contains the XML equivalent of Figure 3.

```
<role id="002" name="Customer"/>
<role id="003" name="Insurance Company"/>
<collaboration id="004" name="Negotiation"/>
<collaboration id="006" name="Contracting"/>
<composition id="035" name="composition">
    <from href="004"/>
    <to href="002"/>
</composition>
<composition id="036" name="composition">
    <from href="004"/>
    <to href="003"/>
</composition>
<composition id="041" name="composition">
    <from href="006"/>
    <to href="002"/>
</composition>
<composition id="042" name="composition">
    <from href="006"/>
    <to href="003"/>
</composition>
```

5 Selection and Visualization

The initial XML document contains the concepts and relations based on the ArchiMate metamodel. It does not contain information about which concepts are relevant for which views, nor does it describe how to visualize the concepts. We can use RML rules to formally define selection and visualization schemes, as will be illustrated in the following sections.

5.1 Selection

Within a single view, usually a selection of the entire set of concepts is made. For example, the Business View in our example only contains *roles* and *collaborations* and abstracts from all other related concepts. For this purpose, RML rules have to filter out all unnecessary information from the XML document and thus create a new document that only contains those concepts and relations that are relevant for the view. By writing down the RML rules, we ensure that we do not omit certain concepts, which often happens when this work is being done by hand.

We have created the following "recipe" for selection:

1. add a specific *selection* element to the XML document which is going to contain the selected concepts;
2. iterate over the document and move all relevant concepts into the specific *selection* element;
3. iterate over the document and move all relevant relations into the specific *selection* element;
4. remove all relations within the *selection* element that have one "dangling" end, i.e. that are related at one side to a concept that does not belong to the selection;
5. remove all elements outside the *selection* element.

Note that the step for removing relations with one "dangling" end out of the selection is necessary, because one relation type (e.g. association) can be defined between several different concept types. Nevertheless, this recipe is fairly easy to understand and can be specified by architects themselves for any kind of selection they want to make.

The following RML rule illustrates the way all instances of a specific concept are included in the selection:

```
<div class="rule">
    <div class="antecedent">
        <model>
            <rml-list name="list1"/>
            <collaboration rml-others="other">
                <rml-list name="childs"/>
            </collaboration>
            <rml-list name="list2"/>
            <selection>
                <rml-list name="selection"/>
            </selection>
            <rml-list name="list3"/>
        </model>
    </div>
    <div class="consequence">
        <model>
            <rml-use name="list1"/>
            <rml-use name="list2"/>
            <rml-use name="list3"/>
            <selection>
                <rml-use name="selection"/>
                <collaboration rml-others="other">
                    <rml-use name="childs"/>
                </collaboration>
            </selection>
        </model>
    </div>
</div>
```

5.2 Visualization

One of the research questions is about the creation of a “dumb” model viewer, i.e., it is ignorant of any architectural language. By this, we can illustrate the way in which XML transformations can be used for creating several visualizations for a single XML document.

For this purpose, we made a specific XML schema which can be interpreted by a model viewer without having to know anything about the ArchiMate language. The following XML fragment illustrates this language.

```
<container height="80" id="014" type="interaction" width="100" >
      <box color="khaki1" height="80" type="round" width="100" x="0" y="0" z="0" />
      <label fieldname="name" halign="center" text="register policy" x="50" y="40" z="1" />
      <icon height="15" type="splitcircle" width="15" x="75" y="10" z="1" />
</container>

<container height="80" id="013" type="interaction" width="100" >
      <box color="khaki1" height="80" type="round" width="100" x="0" y="0" z="0" />
      <label fieldname="name" halign="center" text="sign contract" x="50" y="40" z="1" />
      <icon height="15" type="splitcircle" width="15" x="75" y="10" z="1" />
</container>

<arrow from="013" id="020" to="014" type="triggering" >
      <line type="solid" width="1" z="0" />
      <headarrowtip size="10" type="filledarrow" z="1" />
</arrow>
```

The intermediate visualization language has two main constructs: containers and arrows.

Containers are rectangular areas in which several visual elements can be placed. The exact location of those visual elements can be defined relative to the size and position of the container. Each container has a unique identifier which can be used to refer to the original elements in the architectural description.

Arrows are linear directed elements. They have a head and a tail, which both have to be connected to containers (via their identifiers). They also have unique identifiers themselves.

In the example above, two containers and one arrow are defined. In Figure 7 the output of the interpretation of this XML fragment by the model viewer is shown. As can be seen in the XML fragment, some visual elements, like “split circle”, are built into the model viewer. This has mainly been done for reasons of efficiency.

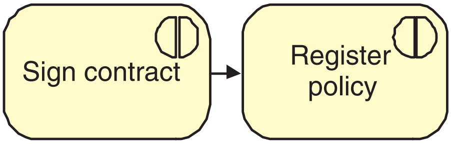


Fig. 7. Example of the visualization technique used

For the transformation of the original XML model to the visualization information, we have created scripts that transform each concept into its corresponding visualization. An example is given below. This example rule transforms an *interaction* concept into a visual representation.

```
<div class="rule">
    <div class="antecedent">
        <interaction id="rml-id" name="rml-name" color="rml-color"/>
    </div>
    <div class="consequence">
        <container id="rml-id" type="interaction" width="100" height="80" color="rml-color">
            <box x="0" y="0" z="0" width="100" height="80" color="khaki1" type="round"/>
            <label x="50" y="40" z="1" halign="center" text="rml-name" fieldname="name"/>
            <icon x="75" y="10" z="1" width="15" height="15" type="splitcircle"/>
        </container>
    </div>
</div>
```

The technique presented here is quite powerful yet easy to understand: from the same architectural description, it is possible to define different visualization styles, like ArchiMate (which is used in the running example), UML[2], etc. In the context of enterprise architectures, this is especially useful since architects often want to have their own style of visualization (for cultural and communication reasons within organizations), without having to conform to a standard defined outside the organization. They can do this in a formal way by capturing their visualizations in XML transformation rules.

6 Analysis

Next to selection and visualization, we investigated ways to use XML transformations for analysis of enterprise architectures. Our aim is to create a technique for impact analysis, i.e., given an entity within the architectural description which is considered to change, which other entities are possibly influenced by this change?

We have created the following recipe for this analysis:

1. add a specific *selection* element to the XML document which is going to contain the concepts that are considered to be possibly influenced;
2. add a special attribute to the element describing the entity under consideration, which can be used for, e.g., visualisation (in order to make it have a red color, for example);
3. make the element describing the entity under consideration a child of the *selection* element;
4. iterate over all relations included in the analysis and, if appropriate, add a special attribute to them and make them a child of the *selection* element;
5. iterate over all concepts and, if appropriate, add a special attribute to them and make them a child of the *selection* element;
6. repeat the previous two steps until the output is stable;
7. remove the *selection* element, so that we have one list of concepts and relations, of which some concepts have a special attribute which indicates that the change possibly has impact on them.

An example of the output of the analysis is given below. The component "InterMed" is considered to change. It has two new attributes. The *selected* attribute indicates that it belongs to the entities which are possibly influenced by the change, while the *special* attribute indicates that this entity is the unique entity considered to change. The remaining elements describe concepts and relations that are all selected, because they are directly or indirectly related to the "InterMed" component.

```
<component id="082" name="InterMed" selected="yes" special="yes"/>

<composition id="104" name="composition" selected="yes" >
    <from href="082" />
    <to href="094" />
</composition>

<interface id="094" name="Interface" selected="yes" />

<assignment id="112" name="assignment" selected="yes" >
    <from href="094" />
    <to href="090" />
</assignment>

<service id="090" name="edit requests"  selected="yes" />
```

Within Figure 8 and Figure 9, the output of the model viewer is given for two views. The change of color is done by the visualization rules, based on the attributes added during the analysis.

By performing impact analysis in the way presented above, we treat the set of architectural concepts as a graph with nodes and edges and define that a node A has impact on another node B when A is selected and there is a relation from A to B. But we could easily change our interpretation of the term "impact", for example by introducing several kinds of impact (e.g., extend, replace, delete) or treating different kind of relations between nodes in a different way. In all

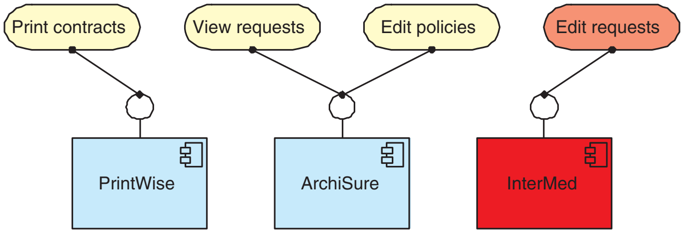


Fig. 8. The Application View with a selected InterMed application

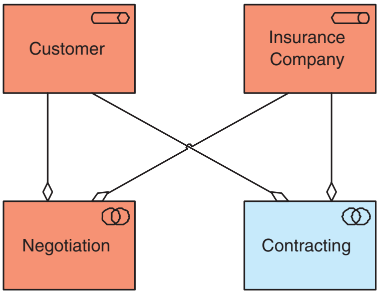


Fig. 9. The resulting Business View after the impact analysis

cases, our interpretation of the term "impact" is formally defined in the XML transformation rules.

7 Conclusions

The research reported on in this paper shows promising results. The use of XML transformation techniques for selection, visualization and analysis of enterprise architectures has several benefits: XML is well-known, the transformation techniques are generic and tools for it are improving rapidly. Transformation rules are well understandable and can be adapted quickly for specific needs or purposes. The application of XML transformation techniques for enterprise architectures is a good example of the way in which formal techniques, in this case, formally defined transformation rules, can be applied effectively by end users in the context of enterprise architecture, i.e. the enterprise architects themselves.

Based on the reported investigations, our answers to the research questions set out earlier are the following:

Question 1. Given a set of architectural information described in a single XML document. How can we use XML transformations to select a subset of this information for a specific architectural view?

This can be done via transformation rules which filter out certain concepts and create a new XML document containing a selection out of the original document. These transformation rules are easy to understand and can be defined by the architects themselves to create their own selections.

Question 2. How can we transform an XML document containing architectural information into another XML document containing visual information in terms of boxes, lines, etc.? How can we build a *model viewer* which interprets this visual information without having to know anything about the architectural language used?

What is needed for this, is a separate "intermediate" language for visualization information. Via XML transformations, we can transform an ArchiMate XML document into an XML document containing only visual elements. The latter document can then be interpreted by a model viewer which only has to know how to interpret the visual elements in this document. By separating the visualization step from the viewer, architects gain much often demanded flexibility: they are able to create their own visualizations in a formal way, i.e. defined in transformation rules.

Question 3. How can we use XML transformations to perform our specific form of impact analysis?

We can do this analysis by iterative selection of the elements which have a relation with the "element to be changed". By including or excluding certain relation types, architects gain insight in the mutual dependencies between the entities within an architecture.

Acknowledgments. This paper is based on research in the context of the ArchiMate[1] project, a research initiative that aims to provide concepts and techniques to support architects in the visualization and analysis of integrated architectures. The ArchiMate consortium consists of ABN AMRO, Stichting Pensioenfonds ABP, the Dutch Tax and Customs Administration, Ordina, Telematica Institute, CWI, University of Nijmegen, and LIACS.

References

1. A. Berlea and H. Seidl. fxt a transformation language for XML documents. *Journal of Computing and Information Technology*, 10(1):19–35, 2002.
2. G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language Reference Manual.* Addison Wesley, 1999.
3. The Rule Markup Initiative community. URL: `http://www.dfki.uni-kl.de/ruleml/`.
4. World Wide Web Consortium. Extensible markup language (XML). URL: `http://www.w3.org/XML/`.
5. World Wide Web Consortium. XSL transformations (XSLT) version 1.0, W3C recommendation, November 1999. URL: `http://www.w3c.org/TR/xslt`.

[1] `http://archimate.telin.nl`

6. J. Jacob. The rule markup language: a tutorial. URL: `http://homepages.cwi.nl/~jacob/rml`
7. H. Jonkers, R. van Buuren, F. Arbab, F.S. de Boer, M.M. Bonsangue, H. Bosma, H. ter Doest, L. Groenewegen, J. Guillen-Scholten, S. Hoppenbrouwers, M. Jacob, W. Janssen, M. Lankhorst, D. van Leeuwen, E. Proper, A. Stam, L. van der Torre, and G. Veldhuijzen van Zanten. Towards a language for coherent enterprise architecture description. In M. Steen and B.R. Bryant, editors, *Proceedings 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003)*, pages 28–39. IEEE Computer Society Press, 2003.
8. James McGovern, Scott W. Ambler, Michael E. Stevens, James Linn, Vikas Sharan, and Elias K. Jo. *A Practical Guide to Enterprise Architecture*. Prentice Hall PTR, 2003.
9. M. Pettersson. RML - a new language and implementation for natural semantics. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming, PLILP*, volume 884 of *LNCS*, pages 117–131. Springer-Verlag, 1994.
10. The Archimate Project. URL: `http://www.telin.nl/NetworkedBusiness/Archimate/ENindex.htm`.
11. IEEE Computer Society. IEEE std 1471-2000: IEEE recommended practice for architectural description of software-intensive systems, Oct. 9, 2000.

Classification and Utilization of Abstractions for Optimization

Dan Quinlan, Markus Schordan,
Qing Yi, and Andreas Saebjornsen

[1] Lawrence Livermore National Laboratory, USA
{dquinlan, yi4}@llnl.gov
[2] Vienna University of Technology
markus@complang.tuwien.ac.at
[3] University of Oslo, Norway
andreas.sabjornsen@fys.uio.no

Abstract. We define a novel approach for optimizing the use of libraries within applications. We propose that library-defined abstractions be annotated with additional semantics to support their automated optimization. By leveraging these additional semantics we enable specialized optimizations of application codes which use library abstractions. We believe that such an approach entails the use of formal methods.

It is a common perception that performance is inversely proportional to the level of abstraction. Our work shows that this is not the case if the additional semantics of library-defined abstractions can be leveraged. We describe ROSE, a framework for building source-to-source translators that perform high-level optimizations on scientific applications. ROSE allows the recognition of library abstractions and the optimization of their use in applications. We show how ROSE can utilize the semantics of user-defined abstractions in libraries within the compile-time optimization of applications.

1 Introduction

User-defined abstractions help software developers be more productive by encapsulating redundant details. Unfortunately the use of these abstractions usually introduces a penalty in performance due to insufficient optimizations from the compilers. We define a performance penalty as the difference between the performance of the direct use of the user-defined abstraction and any semantically equivalent lower level representation. In order to apply optimizations to user-defined abstractions it is often required that these abstractions satisfy certain properties (e.g. operators are side-effect free, array elements are not aliased, etc.). When the compiler cannot verify such properties using program analysis it conservatively disables many performance optimizations. Manually introducing optimization avoid the performance penalty but significantly lower programmer productivity and tie the application to individual computer architectures. Our approach aims at fully automating such optimizations.

T. Margaria and B. Steffen (Eds.): ISoLA 2004, LNCS 4313, pp. 57–73, 2006.

We propose to use annotations to bridge the gap between the semantics implied by the compiler and the additional semantics known only by the library developer who implements the user-defined abstractions. We present a compile-time approach for the optimization of scientific applications using the semantics of library abstractions. We use an array abstraction from the A++/P++ array class library [1] as an example to demonstrate our approach toward the general annotation of user-defined abstractions.

Where we have defined a mechanism to optimize the use of a category of abstractions (array abstractions), our goal is to define a more general mechanism to annotate arbitrary abstractions with semantics supporting their optimizations. We have addressed numerous aspects of the more general problem within the current work. While current work supports only the manual introduction of annotations, future work will explore ways to automate the introduction of such annotations. This paper describes work that has been done to develop an annotation based approach to guide the automated optimization of user-defined abstractions, starting with a motivating example.

1.1 Motivating Example

Figure 1 shows an example of a high-level abstraction from the A++/P++ library [1]. It is an array abstraction which is used within current scientific applications and forms an appropriate target for optimization.

The class `Range` defines a one dimensional iteration space with fixed stride. The class `floatArray` defines an array abstraction where `Range` objects serve as parameters to several operators (e.g `+`, `=`, `()`) for iterating over an array. In the example main function, we show how these classes can be used to compute a new value for each point in the array. Using such abstractions programmers can express complex computations without explicitly introducing loops which permits greater flexibility for optimization (e.g. parallelization, layout optimization, communication optimization, etc.). In the example we use a two-dimensional array with type double as element type.

In figure 1, the function `main` consists of four lines. In line #2 two non-aliased `floatArrays` A and B, each of size 100x100, are created. In line #3, two `Range` objects I and J, defining ranges from 1 to 98 with stride 1 are created. In line #4 the computation is defined. For each point within the specified ranges, a new value is computed on the right hand side as the sum of the four neighbors in horizontal and vertical direction. In general, an arbitrary expression can be specified on the right hand side, in particular using the operators available in the class `floatArray`. In this simple example we have restricted the available functions to `+` and `sin`. The full class consists of about 80 different operators.

In order to optimize this computation the compiler must know both the the aliasing patterns between different array objects and the semantics of array operations.We shall show how such properties can be specified using our annotation language in section 4.

```
class Range {
public:
    Range ( int base, int bound, int stride );
    Range operator+ ( int i );
    Range operator- ( int i );
};
class floatArray {
public:
    floatArray ( int i, int j );
    floatArray & operator= ( const floatArray & X );
    friend floatArray operator+ ( const floatArray & X, const floatArray & Y );
    floatArray operator() ( const Range & I, const Range & J );
    friend floatArray & sin ( const floatArray & X );
};
int main () {                                                /* #1 */
    floatArray A(100,100), B(100,100);                       /* #2 */
    Range I(1,98,1), J(1,98,1);                              /* #3 */
    A(I,J) = B(I-1,J) + B(I+1,J) + B(I,J-1) + B(I,J+1);   /* #4 */
}
```

Fig. 1. Example: Code fragment of header file and program to be optimized

In the remaining sections we present the ROSE architecture [2, 3] used for implementing the presented approach, the annotation based mechanism for the optimization of array abstractions and performance results showing the significant potential for such optimizations.

2 Architecture

The ROSE infrastructure offers several components to build a source-to-source optimizer. A complete C++ front end is available that generates an object-oriented annotated abstract syntax tree (AST) as an intermediate representation. Optimizations are performed on the AST. Several components can be used to build the mid end: a predefined traversal mechanism, an attribute evaluation mechanism, transformation operators to restructure the AST, and pre-defined optimizations. Support for library annotations is available by analyzing pragmas, comments, or separate annotation files. A C++ back end can be used to unparse the AST and generate C++ code. An overview of the architecture is shown in Fig. 2). Steps 1-7, which can be performed by a ROSE source-to-source optimizer, are described in the following sections.

2.1 Front End

We use the Edison Design Group C++ front end (EDG) [4] to parse C++ programs. The EDG front end generates an AST and performs a full type evaluation of the C++ program. This AST is represented as a C data structure. We translate this data structure into an object-oriented abstract syntax tree, Sage III, based on Sage II and Sage++[5]. Sage III is used by the mid end as intermediate representation. The annotated library interfaces are header files which are included by the application program. The AST passed to the mid end represents the program and all the header files included by the program (see Fig. 2, step 1 and 2).

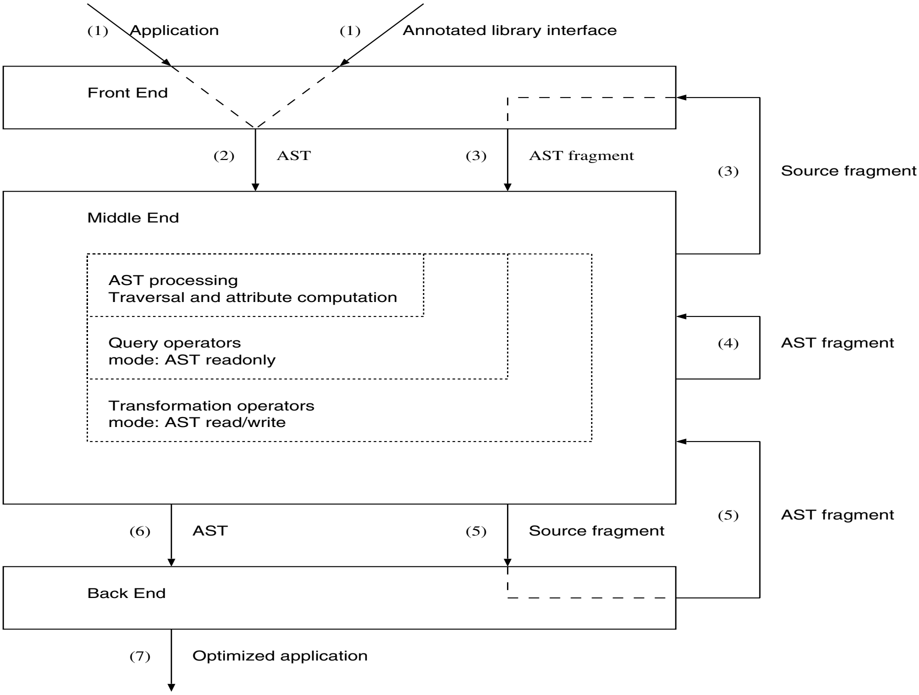


Fig. 2. ROSE Source-To-Source architecture

2.2 Mid End

The mid end permits the restructuring of the AST and performance improving program transformations. Results of program analysis are made available as annotations of AST nodes. The AST processing mechanism computes inherited and synthesized attributes on the AST (see section 2.4 for more details). ROSE also includes a scanner which operates on the token stream of a serialized AST so that parser tools can be used to specify program transformations in semantic actions of an attribute grammar. The grammar is the abstract grammar, generating the set of all ASTs. More details on the use of attribute grammar tools, in particular Coco/R [6] and Frankie Erse's C/C++ port, can be found in [3].

An AST restructuring operation specifies a location in the AST where code should be inserted, deleted, or replaced. Transformation operators can be built using the AST processing mechanism in combination with AST restructuring operations. In Fig. 2 steps 3,4,5 show how the ROSE architecture also uses source code fragments and AST fragments in the specification of program transformations. A fragment is a concrete piece of code or AST. A program transformation is defined by a sequence of AST restructuring operations whereas transformation operators compute such restructuring sequences. Transformations can be parameterized to define conditional restructuring sequences. This is discussed in detail in section 3.

2.3 Back End

The back end unparses the AST and generates C++ source code (see Fig. 2, steps 6 and 7). It can be specified to unparse either all included (header) files or only the source file(s) specified on the command line. This feature is important when transforming user-defined data types, for example, when adding generated methods. Comments are attached to AST nodes and unparsed by the back end.

2.4 AST Processing

The AST processing components permit traversing the AST and computing attributes for each node of the AST. The computed values of attributes can be attached to AST nodes as annotations and used in subsequent optimizations. Context information can be passed down the AST as inherited attributes and results of computations on a subtree can be computed as synthesized attributes (passing information upwards in the tree). Examples for values of inherited and synthesized attributes are the nesting level of loops, the scopes of associated pragma statements, etc. These annotations can be used in transformations to decide whether a restructuring operation can be applied safely. AST processing is used by query operators and transformation operators to compute information according to the structure of the AST and can also be based on annotations computed by other operators. Using this mechanism library developers can build complex high-level transformation operators from lower-level transformation operators.

2.5 AST Query Operators

Building on top of the methods in section 2.4, AST query operators are provided that perform numerous types of predefined queries on the AST. AST query operators may be composed to define complex queries. This mechanism hides some of the details of the AST traversal and is simple and extensible.

2.6 AST Annotations

Source code annotations are represented within the AST as persistent attributes, which can be directly attached to AST nodes. Such attributes can be accessed by subsequent AST processing steps and permit the communication between different AST processing phases. Annotations within the AST include type information obtained from the EDG front end and user-defined attributes, which specify additional semantic information. The additional information can be utilized in transformations to decide whether a restructuring operation is applicable. Annotations can be introduced using several mechanisms supported within ROSE: pragmas, comments, and a separate annotation file mechanism.

3 Transformation Operators

An optimization requires program analysis to determine whether the AST can be restructured such that the semantics of the program are preserved. For the analysis, the AST processing mechanism allows the computation of attributes. A transformation operator includes a predicate and a sequence of AST restructuring operators. The predicate uses the results of the program analysis and additionally the information provided by annotations. The predicate has to ensure that the restructuring sequence can be applied safely. Therefore we combine conservative program analysis with the additional information from annotations to ensure its correctness. Additionally we also provide operations that enforce a certain transformation, such as the *inline* annotation.

Only if the predicate is true the restructuring operators are applied. The sequence of AST restructuring operations can be computed as attributes by the AST processing mechanism or by using an attribute grammar tool, as demonstrated in [3]. Bottom Up Rewrite Systems (BURS), such as burg [7], can be used to operate on the AST. The AST is implemented such that for each node a unique number is available which can be used as operator identifier by such tools. The opportunity to choose between traversals, the AST processing mechanism, attribute grammar tools, or BURS tools allows selection of the most comprehensive specification of a transformation.

A restructuring sequence consists of fragment operators, and as operands AST fragments (subtrees), strings (concrete pieces of code), or AST locations (denoting nodes in the AST).

3.1 Fragment Operators

A fragment operator transforms a given fragment. It permits performing a basic restructuring operation such as insert, delete, or replace on an AST. The target location of an operation in the AST can be absolute or relative (to another absolute location).

A valid fragment can be specified as source fragment or AST fragment. Whether a source fragment is valid with respect to an absolute location is determined automatically. From the syntactic context of the absolute location the prefix, $s_{\triangleright}$, is computed such that all declarations, opening scopes, and function headers are included in the prefix. The postfix, $s_{\triangleright}$, consists of all the syntactic entities of closing scopes (for nested scopes such as for-loops, while-loops, function definitions, etc.).

Definition 1 (Valid Source Fragment). *A source fragment $s_{\Box}$ is* valid with respect to an absolute location l_{abs} *in an AST if it can be completed to a legal program from the syntactic and semantic context of the absolute location such that the completed program has correct syntax and semantics, symb. $s_{\Box}$ is valid with respect to l_{abs} if $frontend(s + s_{\Box} + s_{\triangleright})$ with $s = prefix(l_{abs})$, $s_{\triangleright} = postfix(l_{abs})$ succeeds.*

For example, if we want to insert a source fragment into the AST, we specify a target location as an absolute location, l_{abs}, and from that target location the prefix and postfix can be computed to complete the source fragment to a legal program. Then we can invoke the frontend to obtain the translated AST fragment. Thus, for a valid source fragment $s_{\Box}$ we can always generate a corresponding AST fragment, $ast_{\Box}$. The translation of fragments is an extensive operation. For example, in our C++ infrastructure the AST fragment $ast_{\Box}$ has all templates instantiated and types are determined for all expressions.

Operator	Description
insert : $L_{rel} \times L_{abs} \times ASTs \rightarrow ASTs$	Insertion of AST fragment at relative location (step 4 in Fig. 2)
delete : $L_{abs} \times ASTs \rightarrow ASTs$	Deletion of AST subtree at absolute location in AST (step 4 in Fig. 2)
fragment-frontend : $L_{abs} \times ASTs \times S \rightarrow ASTs$	Translate source fragment with respect to absolute location in AST to corresponding AST fragment (steps 3,5 in Fig. 2)
fragment-backend : $L_{abs} \times ASTs \rightarrow S$	Unparse AST fragment at absolute location in AST to source fragment (step 5 in Fig. 2)
locate : $L_{rel} \times L_{abs} \times ASTs \rightarrow L_{abs}$	Map relative location with respect to absolute location in AST to absolute location in same AST
replace : $L_{rel} \times L_{abs} \times ASTs \times ASTs \rightarrow ASTs$	Replacement of AST fragment at relative location (step 4 in Fig. 2)
replace : $L_{abs} \times ASTs \times S \rightarrow ASTs$	Replacement of AST subtree at absolute location in AST by AST fragment corresponding to source fragment (steps 3,4,5 in Fig. 2)

Fig. 3. Fragment operators which allow to modify the AST by using a relative location, an AST fragment, or a source fragment. Transformation operators are defined as sequence of fragment operations.

In Fig. 3 an overview of the most important fragment operators is given. Based on the handling of code fragments, the transformation operators can be defined as follows. Let $ASTs$ denote the set of ASTs, L_{rel} the set of relative locations in an AST, L_{abs} the set of absolute locations, i.e. the nodes in an AST, and S the set of valid source fragments with respect to an absolute location in the AST. The fragment operators allow rewriting the AST by specifying absolute or relative target locations. A relative location l_{rel} allows the specification of a target location in an AST relative to an absolute location l_{abs}. The operator *locate* maps a relative location l_{rel} to an absolute location l_{abs} in a given AST. Relative locations are used to simplify the specification of the target location of a fragment operation. For example, if a statement can be hoisted out of a loop it suffices to specify as target location the "statement in outer scope right before the current loop". The location "current loop" is specified as absolute location parameter to *locate*. We have defined several classifications of such relative target locations which are useful in making transformations more compact. The insert-operation is an example of using a relative target location. The operator *fragment-frontend* allows translation of source fragments to AST fragments as explained above. It also requires step 5 to compute the necessary prefix and postfix to complete the source fragment to eventually call the front end for the completed program. The unparsing of an AST fragment, *fragment-backend* requires invoking the back end. The last operator listed in Fig. 3, *replace*, allows specification of the new AST fragment, *ast*, which replaces an AST subtree at location L_{abs} in this AST, to be specified by a source fragment, s. This requires all three steps 3,4,5 (see Fig. 2). Step 5 is required to unparse parts of the AST to form the prefix, s, and postfix, $s_{\triangleright}$. In Step 3 the completed source fragment is translated to an AST and the corresponding AST fragment, *ast*, is extracted. Step 4 is the actual rewriting of the AST and the replacement of the AST subtree with the new AST fragment is performed. Based on this basic operations on fragments, transformation operators can be defined.

4 Predefined Optimizations

A large set of compiler optimizations, including both reordering transformations such as loop fusion/fission and blocking, and statement level transformations such as redundant expression elimination, can be applied to improve the performance of applications. Most of these optimizations are under certain safety and profitability constraints, which in turn require specific knowledge of the involved operations. However, because user-defined abstractions often introduce function calls with unknown semantics into an application, many of these compiler optimizations are not performed due to the unknown semantics.

In this section we present techniques that extend the applicability of predefined compiler optimizations. By defining an annotation language, which allows programmers to declare that certain abstractions satisfy the extended requirements of predefined compiler optimizations, we provide an open interface for the programmers to communicate with and to control the underlying optimizations. A preliminary version of our annotation language is shown in Figure 4. In the following, we use the annotation examples in Figure 5 to further illustrate the techniques.

4.1 Enabling Transformations

One of the most significant enabling transformations for library abstractions is inlining, which replaces function calls with their implementations within the calling contexts. Suppose the compiler has access to all the source code of a library. Theoretically, inlining the library code could permit the necessary program analysis and thus allow the compiler to discover the semantics of all abstractions, dismissing the concerns for obscure function calls.

However, the current compilation techniques cannot yet fully bridge the gaps between abstraction semantics and their implementation details. Specifically, reading the library code exposes the underlying implementations, but does not readily permit a discovery of the semantics, such as properties of commutativity and associativity. As the result, we complement inlining transformations with semantics annotations which allows library programmers to define the semantics and control the optimizations of their abstractions.

```
<annot> ::= <annot1>
    | <annot1>;<annot>
<annot1> ::= class <cls_annot>
    | operator <op_annot>
<cls_annot> ::= <clsname>:<cls_annot1>;
<cls_annot1>::= <cls_annot2>
    | <cls_annot2> <cls_annot1>
<cls_annot2>::= <arr_annot>
    | inheritable <arr_annot>
    | has-value { <val_def> }
<arr_annot>::= is-array{ <arr_def>}
    | is-array{define{<stmts>}<arr_def>}
<op_annot> ::= <opdecl> : <op_annot1> ;
<op_annot1> ::= <op_annot2>
    | <op_annot2> <op_annot1>
<op_annot2> ::= modify <namelist>
    | new-array (<aliaslist>){<arr_def>}
    | modify-array (<name>) {<arr_def>}
    | restrict-value {<val_def_list>}
    | read <namelist>
    | alias <nameGrouplist>
    | allow-alias <nameGrouplist>
    | inline <expression>
<arr_def> ::= <arr_attr_def>
    | <arr_attr_def> <arr_def>
<arr_attr_def> ::= <arr_attr>=<expression>;
<arr_attr> ::= dim  | len (<param>)
    | elem(<paramlist>)
    | reshape(<paramlist>)
<val_def> ::= <name>; | <name>;<val_def>
    | <name> =  <expression> ;
    | <name> =  <expression> ; <val_def>
```

Fig. 4. Annotation Grammar

In our annotation language, the programmers can not only customize compilers to inline certain function calls, they can also define additional properties of their abstractions in order to enable specific predefined optimizations. As example, the *inline* annotation in Figure 4 is essentially a "semantics inlining" directive for user-defined functions. In our running example, we use it in the function annotations of class floatArray (see Fig. 5). The function "*floatArray::operator()(int)*" is specified to be inlined and is declared as a subscripted access of the current $floatArray$ object.

4.2 Loop Transformations

As modern computers become increasingly complex, compilers often need to extensively reorder the computation structures of applications to achieve high performance. One important class of such optimizations is the set of loop transformation techniques, such as loop blocking, fusion/fission, and interchange, that has long been applied to Fortran scientific applications. Within ROSE, we have implemented several aggressive loop transformations and have extended them for optimizing loops operating on general object-oriented user abstractions.

Traditional Fortran loop transformation frameworks recognize loops operating on Fortran arrays, that is, arrays with indexed element access and with no aliasing between different elements. After computing the dependence relations between iterations of statements, they then reorder the loop iterations when safe and profitable. To extend this framework, we use an array-abstraction interface to communicate with our loop optimizer the semantics of user-defined array abstractions in C++. The array-abstraction interface both recognizes user-defined array abstractions and determines the aliasing relations between array objects.

In Figure 5, the *is-array* annotation declares that the class $floatArray$ has the pre-defined Fortran array semantics. The array can have at most 6 dimensions, with the length of each dimension i obtained by calling member function $getLength(i)$,

```
class floatArray:
    inheritable is-array { dim = 6; len(i) = this.getLength(i); elem(i$x:0:dim-1) = this(i$x);
    reshape(i$x:0:dim-1) = this.resize(i$x); }; has-value { dim; len$x:0,dim-1=this.getLength(x); }
operator floatArray::operator =(const floatArray& that):
    modify {this}; read {that}; alias none;
    modify-array (this) { dim = that.dim; len(i) = that.len(i); elem(i$x:1:dim) = that.elem(i$x); };
operator +(const floatArray& a1,double a2): modify none; read{a1,a2}; alias none;
    new-array () { dim = a1.dim; len(i) = a1.len(i); elem(i$x:1:dim) = a1.elem(i$x)+a2; };
operator floatArray::operator () (const Range& I):
    modify none; read{I}; alias { (result, this) }; restrict-value { this = { dim = 1; };
    result = {dim = 1; len(0) = I.len;}; };
    new-array (this) { dim = 1; len(0) = I.len; elem(i) = this.elem(i*I.stride + I.base); };
operator floatArray::operator() (int index) : inline { this.elem(index) };
    restrict-value { this = { dim = 1; };};
class Range: has-value { stride; base; len; };
operator Range::Range(int _b,int _l,int _s): modify none; read {_b,_l,_s}; alias none;
    restrict-value { this={base =_b;len=_l;stride=_s;};};
operator + (const Range& lhs, int x ) : modify none; read {lhs,x}; alias none;
    restrict-value { result={stride=lhs.stride; len = lhs.len; base = lhs.base + x; };};
```

Fig. 5. Annotations for classes floatArray and Range

and with each element of the array accessed through the "()" operator. Here the expression $i\$x : 0 : dim - 1$ denotes a list of parameters, i_1,i_2,...,i_{dim-1}. In the grammar the nonterminal <paramlist> corresponds to such expressions. Similarly, the operator "*floatArray::operator= (const floatArray& that)*" is declared to have *modify-array* semantics; that is, it performs element-wise modification of the current array. The operator "+*(const floatArray& a_1, double a_2)*" is declared to have the *new-array* semantics; that is, it constructs a new array with the same shape as that of a_1, and each element of the new array is the result of adding a_2 to the corresponding element of a_1. Similarly, the operator "*floatArray::operator()(const Range& I)*" constructs a new array that is aliased with the current one by selecting only those elements that are within the iteration range I.

Because the safety of loop optimizations is determined by evaluating the side-effects of statements, our annotation language also includes declarations regarding the side-effects of function calls. Specifically, the *mod* annotation declares a list of locations that might be modified by a function call, the *read* annotation declares the list of locations being used, and the *alias* annotation declares the groups of names that might be aliased to each other. These annotations directly communicate with our global alias and side-effect analysis algorithms.

4.3 Statement Level Optimizations

To generate efficient code, most compilers eliminate redundant computations or replace expensive computations with cheaper ones. Commonly used optimizations include constant propagation, constant folding, strength reduction, redundant expression elimination, and dead code elimination. Most of these optimizations

```
void interpolate2D ( floatArray & fineGrid, floatArray & coarseGrid )
{
     int fineGridSizeX   = fineGrid.getLength(0);
     int fineGridSizeY   = fineGrid.getLength(1);
     int coarseGridSizeX = coarseGrid.getLength(0);
     int coarseGridSizeY = coarseGrid.getLength(1);
     // Interior fine grid points
     Range If (2,fineGridSizeX-2,2);  Range Jf (2,fineGridSizeY-2,2);
     Range Ic (1,coarseGridSizeX,1);  Range Jc (1,coarseGridSizeY-1,1);
     // separate loops to be fused
     fineGrid(If,Jf)     =  coarseGrid(Ic,Jc);
     fineGrid(If-1,Jf)   = (coarseGrid(Ic-1,Jc) + coarseGrid(Ic,Jc)) / 2.0;
     fineGrid(If,Jf-1)   = (coarseGrid(Ic,Jc-1) + coarseGrid(Ic,Jc)) / 2.0;
     fineGrid(If-1,Jf-1) = (coarseGrid(Ic-1,Jc-1) + coarseGrid(Ic-1,Jc) +
                            coarseGrid(Ic,Jc-1) + coarseGrid(Ic,Jc)) / 4.0;
}
```

Fig. 6. Example: High-Level user code in function interpolate2D using classes floatArray and Range

```
void interpolate2D(class floatArray &fineGrid,class floatArray &coarseGrid)
{
  int _var_9; int _var_8; int _var_7; int _var_6; int _var_5;
  int _var_4; int _var_3; int _var_2; int _var_1; int _var_0;
  int fineGridSizeX = (fineGrid.length(0));
  int fineGridSizeY = (fineGrid.length(1));
  int coarseGridSizeX = (coarseGrid.length(0));
  int coarseGridSizeY = (coarseGrid.length(1));
  // Interior fine grid points
  class Range If(2,_var_2 = fineGridSizeX - 2,2);
  class Range Jf(2,_var_3 = fineGridSizeY - 2,2);
  class Range Ic(1,coarseGridSizeX,1); class Range Jc(1,coarseGridSizeY - 1,1);
  for (_var_1 = 0; _var_1 <= -1 + (_var_3 + -1 * 2 + 1) / 2; _var_1 += 1) {
    for (_var_0 = 0; _var_0 <= -1 + (_var_2 + -1 * 2 + 1) / 2; _var_0 += 1) {
      fineGrid.elem(_var_0 * 2 + 2,_var_1 * 2 + 2)
        = coarseGrid.elem(_var_0 * 1 + 1,_var_1 * 1 + 1);
    }
  }
  for (_var_5 = 0; _var_5 <= -1 + (_var_3 + -1 * 2 + 1) / 2; _var_5 += 1) {
    for (_var_4 = 0; _var_4 <= -1 + (_var_2 + -1 * 2 + 1) / 2; _var_4 += 1) {
      fineGrid.elem(2 + -1 * 1 + _var_4 * 2,_var_5 * 2 + 2)
        = (coarseGrid.elem(1 + -1 * 1 + _var_4 * 1,_var_5 * 1 + 1)
        + coarseGrid.elem(_var_4 * 1 + 1,_var_5 * 1 + 1)) / 2.0;
    }
  }
  for (_var_7 = 0; _var_7 <= -1 + (_var_3 + -1 * 2 + 1) / 2; _var_7 += 1) {
    for (_var_6 = 0; _var_6 <= -1 + (_var_2 + -1 * 2 + 1) / 2; _var_6 += 1) {
      fineGrid.elem(_var_6 * 2 + 2,2 + -1 * 1 + _var_7 * 2)
        = (coarseGrid.elem(_var_6 * 1 + 1,1 + -1 * 1 + _var_7 * 1)
        + coarseGrid.elem(_var_6 * 1 + 1,_var_7 * 1 + 1)) / 2.0;
    }
  }
  for (_var_9 = 0; _var_9 <= -1 + (_var_3 + -1 * 2 + 1) / 2; _var_9 += 1) {
    for (_var_8 = 0; _var_8 <= -1 + (_var_2 + -1 * 2 + 1) / 2; _var_8 += 1) {
      fineGrid.elem(2 + -1 * 1 + _var_8 * 2,2 + -1 * 1 + _var_9 * 2)
        = (coarseGrid.elem(1 + -1 * 1 + _var_8 * 1,1 + -1 * 1 + _var_9 * 1)
        + coarseGrid.elem(1 + -1 * 1 + _var_8 * 1,_var_9 * 1 + 1)
        + coarseGrid.elem(_var_8 * 1 + 1,1 + -1 * 1 + _var_9 * 1)
        + coarseGrid.elem(_var_8 * 1 + 1,_var_9 * 1 + 1)) / 4.0;
    }
  }
}
```

Fig. 7. Example: Function interpolate2D with translated array operations only

```
void interpolate2D(class floatArray &fineGrid,class floatArray &coarseGrid)
{
  int coarseGrid_length_2 = (coarseGrid.Array_Descriptor.Array_Domain.getLength(1));
  int coarseGrid_length_1 = (coarseGrid.Array_Descriptor.Array_Domain.getLength(0));
  int coarseGrid_stride_2 = (coarseGrid.Array_Descriptor.Array_Domain.Stride[1]);
  int coarseGrid_stride_1 = (coarseGrid.Array_Descriptor.Array_Domain.Stride[0]);
  int coarseGrid_size_2 = (coarseGrid.Array_Descriptor.Array_Domain.Size[1]);
  int coarseGrid_size_1 = (coarseGrid.Array_Descriptor.Array_Domain.Size[0]);
  float *coarseGrid_pointer = (coarseGrid.getDataPointer());
  int fineGrid_length_2 = (fineGrid.Array_Descriptor.Array_Domain.getLength(1));
  int fineGrid_length_1 = (fineGrid.Array_Descriptor.Array_Domain.getLength(0));
  int fineGrid_stride_2 = (fineGrid.Array_Descriptor.Array_Domain.Stride[1]);
  int fineGrid_stride_1 = (fineGrid.Array_Descriptor.Array_Domain.Stride[0]);
  int fineGrid_size_2 = (fineGrid.Array_Descriptor.Array_Domain.Size[1]);
  int fineGrid_size_1 = (fineGrid.Array_Descriptor.Array_Domain.Size[0]);
  float *fineGrid_pointer = (fineGrid.getDataPointer());
  int _var_9; int _var_8; int _var_7; int _var_6; int _var_5;
  int _var_4; int _var_3; int _var_2; int _var_1; int _var_0;
  int fineGridSizeX = (fineGrid_length_1);
  int fineGridSizeY = (fineGrid_length_2);
  int coarseGridSizeX = (coarseGrid_length_1);
  int coarseGridSizeY = (coarseGrid_length_2);

  // Interior fine grid points
  class Range If(2,_var_2 = fineGridSizeX - 2,2);
  class Range Jf(2,_var_3 = fineGridSizeY - 2,2);
  class Range Ic(1,coarseGridSizeX,1);
  class Range Jc(1,coarseGridSizeY - 1,1);

  for (_var_1 = 0; _var_1 <= -1 + (_var_3 + -1) / 2; _var_1 += 1) {
    for (_var_0 = 0; _var_0 <= -1 + (_var_2 + -1) / 2; _var_0 += 1) {
      fineGrid_pointer[_var_0 * 2 + 2 + (_var_1 * 2 + 2)
                       * fineGrid_stride_1 * fineGrid_size_1]
        = coarseGrid_pointer[_var_0 * 1 + 1 + (_var_1 * 1 + 1)
                             * coarseGrid_stride_1 * coarseGrid_size_1];
      fineGrid_pointer[2 + -1 * 1 + _var_0 * 2 + (_var_1 * 2 + 2)
                       * fineGrid_stride_1 * fineGrid_size_1]
        = (coarseGrid_pointer[1 + -1 * 1 + _var_0 * 1 + (_var_1 * 1 + 1)
                              * coarseGrid_stride_1 * coarseGrid_size_1]
        + coarseGrid_pointer[_var_0 * 1 + 1 + (_var_1 * 1 + 1)
                             * coarseGrid_stride_1 * coarseGrid_size_1]) / 2.0;
      fineGrid_pointer[_var_0 * 2 + 2 + (2 + -1 * 1 + _var_1 * 2)
                       * fineGrid_stride_1 * fineGrid_size_1]
        = (coarseGrid_pointer[_var_0 * 1 + 1 + (1 + -1 * 1 + _var_1 * 1)
                              * coarseGrid_stride_1 * coarseGrid_size_1]
        + coarseGrid_pointer[_var_0 * 1 + 1 + (_var_1 * 1 + 1)
                             * coarseGrid_stride_1 * coarseGrid_size_1]) / 2.0;
      fineGrid_pointer[2 + -1 * 1 + _var_0 * 2 + (2 + -1 * 1 + _var_1 * 2)
                       * fineGrid_stride_1 * fineGrid_size_1]
        = (coarseGrid_pointer[1 + -1 * 1 + _var_0 * 1 + (1 + -1 * 1 + _var_1 * 1)
                              * coarseGrid_stride_1 * coarseGrid_size_1]
        + coarseGrid_pointer[1 + -1 * 1 + _var_0 * 1 + (_var_1 * 1 + 1)
                             * coarseGrid_stride_1 * coarseGrid_size_1]
        + coarseGrid_pointer[_var_0 * 1 + 1 + (1 + -1 * 1 + _var_1 * 1)
                             * coarseGrid_stride_1 * coarseGrid_size_1]
        + coarseGrid_pointer[_var_0 * 1 + 1 + (_var_1 * 1 + 1)
                             * coarseGrid_stride_1 * coarseGrid_size_1]) / 4.0;
    }
  }
}
```

Fig. 8. Example: optimized function interpolate2d (translating array operations + loop fusion)

have been applied only to expressions composed of built-in types. We enable the optimization of statements and expressions of high-level user-defined abstractions. Additional annotations are used to specify the semantics of user-defined types which are not determined by the existing program analysis. The additional semantics are utilized in optimizing user-defined abstractions.

As an example, we have implemented an adapted constant-propagation/folding algorithm to automatically infer the symbolic properties of arbitrary user-defined objects. In Figure 5, two annotations, *has-value* and *restrict-value*, are used to describe the properties. Specifically, *has-value* declares that class *floatArray* has two properties: the array dimension and the length of each dimension i, and that class *Range* has three properties, *base*, *len* and *stride*, for selecting subsets of elements from arrays. Similarly, the annotation *restrict-value* declares how properties of user-defined types can be implied from function calls. For example, if "*floatArray::operator()(int index)*" is used to access the element of an *floatArray* object *arr*, we know that *arr* must have a single dimension, and it will remain single-dimensional until some other operator modifies its shape. We have also combined the symbolic property analysis with loop optimizations to automatically determine the shapes of user-defined abstractions [8].

Using these annotations as additional semantic information we can translate high-level array operations to a sequence of loops and perform loop fusion. For example, in Fig. 6 we show the high-level code of a 2D interpolation. The essential information from annotations is the element-wise modification of the array (modify annotation). The translated source code is shown in Fig. 7. Additionally, we applied loop fusion to this version and obtained an even more efficient version of the program, shown in Fig. 8. Both optimization steps use information from the annotations. In the next section we discuss the performance of these optimized versions and other benchmarks.

5 Experimental Results

This section presents some preliminary results from applying loop optimizations to several kernels written using the A++/P++ Library [1], an array class library that supports both serial and parallel array abstractions with a single interface. We selected our kernels from the Multigrid algorithm for solving elliptic partial differential equations. The Multigrid algorithm consists of three phases: relaxation, restriction, and interpolation, from which we selected both interpolation and relaxation on one, two, and three dimensional problems.

Our experiments aim to validate two conclusions: our approach can significantly improve the performance of numerical applications, and our approach is general enough for optimizing a large class of applications using object-oriented abstractions. The kernels we used, though small, use a real-world array abstraction library and are representative of a much broader class of numerical computations expressed using sequences of array operations. All six kernels (one, two and three-dimensional interpolation and relaxation) benefited significantly from our optimizations.

Table 1. Performance results (*orig*: elapsed time of original versions written using array abstractions — different numbers of iterations were run for different problem sizes; *translate-only*: speedups from translating array abstractions into low-level C implementations; *translate+fusion*: speedups from both array translation and loop fusion; *fusion-only*: speedups from applying loop fusion alone.)

(a) Interpolation results

	Interp1D				Interp2D				Interp3D			
array size	orig (sec)	transla te only	translate + fusion	fusion only	orig (sec)	transla te only	translate + fusion	fusion only	orig (sec)	transla te only	translate + fusion	fusion only
50	4.833	1.915	2.131	1.113	7.000	3.034	3.932	1.296	9.166	2.497	3.184	1.275
75	5.000	4.142	4.519	1.091	7.000	2.766	3.131	1.132	9.333	3.021	3.813	1.262
100	5.333	2.593	2.899	1.118	7.000	2.753	3.247	1.179	9.333	2.929	3.767	1.286
125	7.666	2.853	4.228	1.482	9.833	3.304	3.882	1.175	10.666	3.214	4.442	1.382
150	9.166	2.390	4.214	1.763	11.166	2.897	4.542	1.568	12.333	2.871	4.189	1.459
175	11.366	2.630	4.618	1.756	12.833	2.893	4.964	1.716	15.766	3.403	5.264	1.547
200	11.000	2.419	4.289	1.773	14.799	3.161	5.348	1.692	13.799	2.514	4.211	1.675

(b) Red-black relaxation results

	RedBlack1D				RedBlack2D				RedBlack3D			
array size	orig (sec)	transla te only	translate + fusion	fusion only	orig (sec)	transla te only	translate + fusion	fusion only	orig (sec)	transla te only	translate + fusion	fusion only
50	11.500	2.178	5.338	2.451	17.166	1.650	3.344	2.026	22.499	3.260	3.445	1.057
75	14.999	1.728	6.692	3.872	16.666	1.627	3.280	2.016	27.332	3.938	3.776	0.959
100	26.166	3.540	11.852	3.348	32.165	2.672	5.146	1.926	35.665	4.744	4.176	0.880
125	32.499	1.960	12.327	6.289	41.498	2.418	4.421	1.828	45.998	4.685	3.895	0.831
150	35.165	2.865	13.885	4.847	46.665	2.134	4.643	2.176	53.498	5.272	4.440	0.842
175	38.132	2.344	15.270	6.513	52.065	2.514	5.378	2.140	64.531	6.238	5.701	0.914
200	38.598	3.125	15.117	4.838	53.398	2.501	6.117	2.446	67.797	6.703	5.384	0.803

We generated three versions for each kernel: the original version (*orig*) using array abstractions, the *translate-only* version auto-optimized by translating array operations into low level C implementations, and the *translate+fusion* version auto-optimized both with array translation and loop fusion. For the two-dimensional interpolation code Fig. 6 shows the *original*, Fig. 7 the *translate-only* version, and Fig. 8 the *translate+fusion* version.

The original versions of all kernels each have 20-60 lines of code, (they look simple because they are written using array abstractions). After translating array operations into explicit loops, each kernel contains 2-8 loop nests which are then considered for loop optimization. Each loop nest has 1-3 dimensions, depending on the dimensionality of the arrays being modified.

We measured all versions on a Compaq AlphaServer DS20E. Each node has 4GB memory and two 667MHz processors. Each processor has L1 instruction and data caches of 64KB each, and 8MB L2 cache. We used the Compaq vendor C++ compiler with the highest level of optimization, and measured the elapsed-time of each execution. Table 1 present our measurements using multiple array sizes.

From Table 1(a), in nearly all cases the translation of the array abstractions results in significant improvements. But applying loop fusion improves the performance further by 20%-75%. This validates our belief that loop optimization is a significant step further toward fully recovering the performance penalty of using high-level array abstractions.

From Table 1(b), the dominant performance improvements come from translating array abstractions into low-level implementations(*translate-only*). Loop fusion can further improve performance by 2.3-6.5 times for one and two-dimensional relaxation kernels, but for three-dimensional relaxation, it showed only slight improvement (5%) for small arrays(50) and degraded performance (up to 20%) for large arrays. Here the performance degradation is due to increased register pressures from the much larger fused loop bodies in the three-dimensional case. We are working on better algorithms to selectively apply loop fusion.

The final codes generated by our optimizer are very similar to the corresponding C programs that programmers would manually write. Consequently, we believe that their performance would also be similar. Further, because programmers usually don't go out-of-the-way in applying loop optimizations, our techniques can sometimes perform better than hand-written code. This is especially true for the red-black relaxation kernels, where the original loops need to be re-aligned before fusion and a later loop-splitting step is necessary to remove conditionals inside the fused loop nests. Such complex transformations are much more easily applied automatically by compilers than manually by programmers.

6 Related Work

Related work on the optimization of libraries in telescoping languages [9] shares similar goals as our research. The SUIF compiler [10] and OpenC++ [11] each provided a programmable level of control over the compilation of applications in support of optimizing user-defined abstractions. The Broadway compiler [12] uses general *annotation languages* to guide source code optimizations. Within ROSE, we provide both an open compiler infrastructure for programmers to define their own optimizations and a collection of annotation mechanisms for programmers to exploit predefined traditional compiler optimizations. Template Meta-Programming[13,14] has also been used to optimize user-defined abstractions, but is effective only when optimizations are isolated within a single statement. Optimizations across statements, such as loop fusion, is beyond the capabilities of template meta-programming.

A rich set of compiler optimization techniques have been developed to improve the performance of applications, including a collection of loop transformations. These transformations by default can only optimize operations on primitive types, whose semantics are known by the compilers. To extend these optimizations to user-defined abstractions, Wu, Midkiff, Moreira and Gupta [15] proposed *semantic inlining*, which treats specific user-defined types as primitive types in Java. Artigas, Gupta, Midkiff and Moreira [16] devised an *alias versioning* transformation that creates alias-free regions in Java programs so that loop optimizations can be applied to Java primitive arrays and the array abstractions from their library. Wu and Padua [17] investigated automatically parallelization of loops operating on user-defined containers, but assumed that their compiler knew about the semantics of all operators. All the above approaches apply compiler techniques to optimize library abstractions. However, by encoding the knowledge within their compilers, these specialized compilers cannot be used to optimize abstractions in general other than those in their libraries. In contrast, we target optimizing general user-defined abstractions by allowing programmers to classify their abstractions and to explicitly communicate semantics information with the compiler.

7 Conclusions

User-defined abstractions are productive in the development of application codes, but the abstraction penalty is often not acceptable for scientific computing. We have presented an approach that reduces this penalty such that the performance of user-defined abstractions becomes acceptable for high-performance computing, allowing to use these abstractions to achieve higher productivity in the development of scientific applications.

We have demonstrated that leveraging semantics of user-defined abstractions can provide significant opportunities for our optimizations and identified an annotation approach to specify relevant user-defined semantics. Using these annotations, we built an automated transformation approach greatly simplifying the otherwise explicit specification of program transformations using more traditional approaches (such as the other mechanisms in ROSE).

We expect that additional research work on the classification of general abstractions will lead to a more useful and practical optimization approach tailored to the domain specific optimization opportunities of user-defined abstractions. The success of such an approach depends upon other research areas, including the verification of transformations, verification of annotations, and the semantic classification of general user-defined abstractions.

References

1. R. Parsons and D. Quinlan. A++/P++ array classes for architecture independent finite difference computations. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, April 1994.
2. Daniel Quinlan, Markus Schordan, Brian Miller, and Markus Kowarschik. Parallel object-oriented framework optimization. *Concurrency and Computation: Practice and Experience*, 16, Issue 2-3:293–302, February 2004.
3. Markus Schordan and Daniel Quinlan. A source-to-source architecture for user-defined optimizations. In *JMLC'03: Joint Modular Languages Conference*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer Verlag, August 2003.
4. Edison Design Group. http://www.edg.com.
5. Francois Bodin, Peter Beckman, Dennis Gannon, Jacob Gotwals, Srinivas Narayana, Suresh Srinivas, and Beata Winnicka. Sage++: An object-oriented toolkit and class library for building fortran and C++ restructuring tools. In *Proceedings. OONSKI '94*, Oregon, 1994.
6. Hanspeter Moessenboeck. Coco/R - A generator for production quality compilers. In *LNCS477, Springer*, 1991.
7. Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG: fast optimal instruction selection and tree parsing. *ACM SIGPLAN Notices*, 27(4):68–76, April 1992.

8. Qing Yi and Dan Quinlan. Applying loop optimizations to object-oriented abstractions through general classification of array semantics. In *The 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, Sep 2004.
9. Ken Kennedy, Bradley Broom, Keith Cooper, Jack Dongarra, Rob Fowler, Dennis Gannon, Lennart Johnsson, John Mellor-Crummey, and Linda Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 61(12):1803–1826, December 2001.
10. M. S. Lam S. P. Amarasinghe, J. M. Anderson and C. W. Tseng. The suif compiler for scalable parallel machines. In *in Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, Feb 1995.
11. Shigeru Chiba. Macro processing in object-oriented languages. In *TOOLS Pacific '98, Technology of Object-Oriented Languages and Systems*, 1998.
12. Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. *ACM SIGPLAN Notices*, 35(1):39–52, January 2000.
13. Todd Veldhuizen. Expression templates. In S.B. Lippmann, editor, *C++ Gems*. Prentice-Hall, 1996.
14. Federico Bassetti, Kei Davis, and Dan Quinlan. A comparison of performance-enhancing strategies for parallel numerical object-oriented frameworks. In Ishikawa et al., editor, *International Scientific Computing in Object-Oriented Parallel Environments, ISCOPE 97*, volume 1343 of *LNCS*. Springer, 1997.
15. Peng Wu, Samuel P. Midkiff, Jose E. Moreira, and Manish Gupta. Improving Java performance through semantic inlining. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, Mar 1999.
16. Pedro V. Artigas, Manish Gupta, Samuel Midkiff, and Jose Moreira. Automatic loop transformations and parallelization for Java. In *Proceedings of the 2000 International Conference on Supercomputing*, May 2000.
17. Peng Wu and David Padua. Containers on the parallelization of general-purpose Java programs. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, Oct 1999.

On the Correctness of Transformations in Compiler Back-Ends

Wolf Zimmermann

Institut für Informatik, Martin-Luther-Universität Halle-Wittenberg
06099 Halle/Saale, Germany
zimmer@informatik.uni-halle.de

Abstract. This paper summarizes the results on the correctness of the transformations in compiler back-ends achieved in the DFG-project *Verifix*. Compiler back-ends transform intermediate languages into code of the target machine. Back-end generators allow to generate compiler back-ends from a set of transformation rules. This paper focuses on the correctness of these transformation rules and on the correctness of the whole transformation stemming from the transformation rules.

1 Introduction

Verification of software systems is often done on the level of the source language. However, it is the binary code generated by a compiler that is really executed. Thus, unless the compiler guarantees the correctness of its compilation, a verification of the source code doesn't say anything about the correctness of the binary code. In order to avoid bugs at the binary level, either the binary code has to be verified directly or source code has to be verified and compiled into binary code by a correct compiler. Compiler bugs are more frequent than expected (see e.g. the Borland Pascal compiler Bug List and the Java Bug Database).

The aim of the DFG-project[1] *Verifix* was to develop approaches for the construction of *verifying* compilers. These guarantee that if a target program τ is generated from a source program σ, then τ is a correct translation of σ. The approaches should work for *realistic* source languages, as e.g. defined by ISO-standards, and *realistic* target languages as defined e.g. by assembly languages of industrial processors. Apart from these goals, a verifying compiler should generate code that is as good in the sense of time and space as it could be generated by a non-verifying compiler. *Verifix* achieved these goals by using the classical compiler architecture which is well-established since more than 25 years. In particular, this decision implies that there are no restrictions on the choice of source and target languages. Although not considered as a part of the project, verification shouldn't restrict the use of optimizing transformations.

This paper focuses on the correctness of the transformations in the code generation phase, i.e. how to verify the transformation from intermediate code to binary code. [24,21] show how this part is embedded into a whole verifying

[1] *Verifix* was supported by DFG grants Go 323/3-3, He 2411/2-3, La 426/15-3.

T. Margaria and B. Steffen (Eds.): ISoLA 2004, LNCS 4313, pp. 74–95, 2006.

compiler. We consider classical basic-block oriented intermediate languages and sequential register based target machines. This also includes pipelined and superscalar processors where scheduling is done by hardware. However, we do not consider optimizations on this level. The requirements are the same as used by back-end generators such as BEG [14,13]. They generate compiler back-ends from transformation rules specified as a special class of term-rewrite rules. Code-generation consists of two phases: code selection and assembly. Code selection replaces all instructions except jumps in the basic blocks by machine instructions (in their binary format). Assembly linearizes the basic block graphs and thereby generates jump instructions (again in binary format). Tools such as BEG specify code selection. We finally assume that the formal operational semantics of the intermediate language and the target language is given as Abstract State Machines. The contribution of *Verifix* is that from a practical viewpoint there are just two cases to consider (transformation of side-effect free expressions and instructions with side-effects) and for each of these cases there is a simple mechanizable proof strategy to prove the correctness of the corresponding transformation rule.

The paper is organized as follows: Section 2 introduces Abstract State Machines as required for this paper. Section 3 sketches code selection by term-rewrite systems. Section 4 introduces the notion of correctness and applies it to the correctness of transformation specified by term-rewrite systems. Section 5 discusses the two proof strategies and argues why these two are sufficient. Section 6 introduces the task of assembly, the transformations, and the approach for correctness proofs. Section 7 discusses related work and Section 8 presents our conclusions.

2 Abstract State Machines and Language Semantics

We first define the notion of Abstract State Machines. For more details, we refer to the Lipari-Guide [25] and the ASM 1997 Guide [26]. An *Abstract State Machine* (ASM) is a tuple $(\Sigma, \Phi_{\mathrm{Init}}, \mathit{Trans})$ where Σ is a signature, Φ_{Init} is a set of Σ-formulas (the *initial conditions*), and *Trans* is a finite set of *transition rules*. The set of *states* is the set $\mathrm{Alg}(\Sigma)$ of Σ-algebras. Σ-terms are defined as usual. A state q is *initial* iff q is a model of Φ_{Init} in the sense of logic, denoted as $q \models \Phi_{\mathrm{Init}}$. In this article, we use order-sorted partial Σ-algebras.

Notation: Sorts are denoted by capital letters, function symbols always start with a lower-case letter. $S_1 < S_2$ denotes that S_1 is a sub-sort of S_2. The symbol $f : T_1 \times \ldots \times T_n \to T$ denotes a function symbol representing a total function and $f : T_1 \times \ldots \times T_n \to ?T$ denotes a function symbol representing a partial function. $[\![\cdot]\!]_q$ denotes the interpretation function of symbols of Σ in Σ-algebra q. This notion is extended as usual to Σ-terms. $t \in S$ is satisfied iff t is a term of sort S. $t[x/t']$ denotes the term t where each occurrence of variable x is substituted by term t'. This can be extended as usual to Σ-formulas (here, only free variables are substituted) and to transitions.

Transition rules are specified as in [26]. In this paper, we use the following kinds of transition rules: **if** φ **then** *Transitions* **endif**
if φ **then** *Transitions*$_1$ **else** *Transitions*$_2$ **endif**

Table 1. Typical Instructions of an Intermediate Language

$:=_I$:	$\mathit{EXPR} \times \mathit{EXPR}$	$\rightarrow \mathit{ASSIGN}$	integer assignment
$:=_A$:	$\mathit{EXPR} \times \mathit{EXPR}$	$\rightarrow \mathit{ASSIGN}$	address assignment
goto:	LABEL	$\rightarrow \mathit{JUMP}$	unconditional jump
$+_I$:	$\mathit{EXPR} \times \mathit{EXPR}$	$\rightarrow \mathit{EXPR}$	integer addition
$+_A$:	$\mathit{EXPR} \times \mathit{EXPR}$	$\rightarrow \mathit{EXPR}$	address addition
ld_I:	EXPR	$\rightarrow \mathit{EXPR}$	read integer from memory
ld_A:	EXPR	$\rightarrow \mathit{EXPR}$	read address from memory
int_{ck}:		$\rightarrow \mathit{EXPR}$	k-bit integer constant
addr_{ck}:		$\rightarrow \mathit{EXPR}$	k-bit address constant

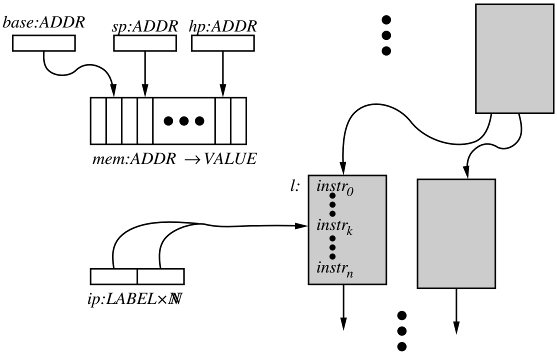


Fig. 1. State Space of an Example Intermediate Language

where φ is formula of many-sorted first order predicate logic and *Transitions* is a set of transitions of one of the above forms or an update $f(t_1,\ldots,t_n) := t$. Such an update changes the interpretation of f: if the state transition from state q to state q' executes an update $f(t_1,\ldots,t_n) := t$ then

$$[\![f]\!]_{q'}(a_1,\ldots,a_n) = \begin{cases} [\![t]\!]_q & \text{if } a_1 = [\![t_1]\!]_q,\ldots,a_n = [\![t_n]\!]_q \\ [\![f]\!]_q(a_1,\ldots,a_n) & \text{otherwise} \end{cases}$$

The first kind of transition rule executes the transition rules of *Transitions* in a state q iff $q \models \varphi$. Similarly, the second kind of transition rule executes the transition rules of $\mathit{Transitions}_1$ in a state q iff $q \models \varphi$. Otherwise, the transition rules of $\mathit{Transitions}_2$ are executed in a state q. Macros can be used to abbreviate updates and terms. It is just a parameterized textual substitution mechanism. A macro definition has the form $\mathit{macro}(\mathit{par}_1,\ldots,\mathit{par}_k) \triangleq \mathit{text}$. If in an ASM-specification a term $\mathit{macro}(\mathit{arg}_1,\ldots,\mathit{arg}_k)$ is used, then it is textually replaced by $\mathit{text}[\mathit{par}_1/\mathit{arg}_1,\ldots,\mathit{par}_k/\mathit{arg}_k]$. A function is *dynamic* iff its interpretation may change, otherwise a function is called *static*. Note that a function may be dynamic because of explicit updates in transition rules or its definition depends on dynamic functions.

Abstract State Machines can be viewed as state transition systems. A *state transition system* is a triple $(Q, I, \rightarrow)$ where Q is the (possibly infinite) set of states, $I \subseteq Q$ is the set of initial states and $\rightarrow \subseteq Q \times Q$ is the state transition relation. For an ASM $(\Sigma, \Phi_{init}, \mathit{Trans})$, it is $Q = \mathrm{Alg}(\Sigma)$ the set of Σ-algebras, $I = \{q \in Q : q \models \Phi_{init}\}$ the set of Σ-algebras satisfying the initial conditions, and *Trans* describes the state transition relation as above.

Example 1. The main principles for designing a dynamic semantics of programming languages are: First, define the (abstract) syntax and the state space and then define the transition rules for each class of instructions. Here, we only give a fragment for an intermediate language based on basic blocks. A *program* π is a set of procedures, a *procedure* p is a basic block graph, i.e., a labeled directed graph (BB, E, lab) with a designated initial basic block i where $lab : BB \to LABEL$, $LABEL$ is the sort of symbolic labels, each *basic block* $bb \in BB$ is a finite sequence of instructions containing no *jump*-instruction except possibly the last instruction, and there is an edge $(bb_1, bb_2) \in E$ iff the jump instruction of bb_1 has the jump target $lab(bb_2)$. For simplicity, we assume that each basic block in a program has a unique label. Note that this implies that the inverse function $lab^{-1} : LABEL \to BB$ is well-defined. Table 1 shows the signature of some typical instructions. Each instruction is a term over this signature and can therefore be viewed as a tree.

The state space (see Fig. 1) consists of the memory *mem* of the target machine, $ADDR$ and $VALUE$ are the sort of addresses and the sort of values that can be stored in the memory, a stack pointer *sp*, a base address *base*, a heap pointer *hp*, and an instruction pointer *ip*. $ip = (l, k)$ means that the instruction to be executed next is the k-th instruction of the basic block bb with label l. Note that memory mapping is often part of the intermediate code generation. In particular the execution environment is already mapped into memory.

Fig. 2 shows the definitions of expression evaluation, the ASM state transition rules for integer assignment, and the state transition rules for jumps. Note that *eval* is a dynamic function. The function $SExt_k(x)$ is a signed extension of the k-bit sequence x to a 64-bit integer or a 64-bit relative address[2]. The function *instr* computes the instruction corresponding to an instruction pointer. The macro *ip* **is** T defines the class of instruction, the macro *Proceed* advances the instruction pointer if it has not yet reached the end of a basic block (k-tuples are denoted as $(t_1, \ldots, t_k)$ and the projection to the i-th element of a tuple t is denoted by $t \downarrow_i$), the functions $lhs, rhs : LABEL \times \mathbb{N} \to ?EXPR$ compute the left and right hand sides of an assignment, respectively, and the function $target : LABEL \times \mathbb{N} \to ?LABEL$ computes the jump target.

The next example contains some state transition rules from the DEC-Alpha machine language specification, for a complete specification see [16].

Example 2. All necessary information for the formal semantics of machine languages is provided by the processor manual. Usually, the dynamic semantics is given by register transfers which already are very close to updates of ASMs. The semantics can immediately be provided for the binary format of instructions

[2] Here, we use the DEC-Alpha as target machine which has 64-bit integer and address arithmetic. If a 32-bit machine is used then the definition of the intermediate language has to be changed accordingly.

$$
\begin{aligned}
&\mathit{eval} : \mathit{EXPR} \to \mathit{VALUE}\\
&\mathit{eval}(\mathit{int}_{ck}) = \mathit{SExt}_k(c)\\
&\mathit{eval}(\mathit{addr}_{ck}) = \mathit{base} \oplus_A \mathit{SExt}_k(c)\\
&\mathit{eval}(\mathit{ld}_I(x)) = \mathit{mem}(\mathit{eval}(x))\\
&\mathit{eval}(\mathit{ld}_A(x)) = \mathit{mem}(\mathit{eval}(x))\\
&\mathit{eval}(x +_I y) = \mathit{eval}(x) \oplus_I \mathit{eval}(y)\\
&\mathit{eval}(x +_A y) = \mathit{eval}(x) \oplus_A \mathit{eval}(y)
\end{aligned}
$$

$\oplus_I, \oplus_A$ is integer/address addition on the target machine

$$\mathit{instr}((l,k)) = \mathit{lab}^{-1}(l)_k$$

```
if $ip$ is $ASSIGN$ then
    $mem(eval(lhs(ip))) := eval(rhs(ip))$
    $Proceed$
endif
if $ip$ is $JMP$ then
    $ip := (target(ip), 0)$
endif
```

Macros:
ip **is** $T \triangleq \mathit{instr}(ip) \in T$
$\mathit{Proceed} \triangleq ip := (ip\downarrow_1, ip\downarrow_2 +1)$

Fig. 2. Expression Evaluation and State Transitions

using adequate access functions, cf. Fig. 3. They use sorts of bit sequences of a certain length: *QUAD*, *LONG*, *WORD*, *TFCODE*, *BYTE*, *OPCODE*, and *RADDR* denote the sorts of 64-bit sequences (quad integers and addresses), 32-bit sequences (long integers), 16-bit sequences (words), 11-bit sequences (type and function code[3]), 8-bit sequences (bytes), 6-bit sequences (operation codes), and 5-bit sequences (register addresses), respectively. Bit sequences may denoted in hexadecimal, decimal, binary notation (as in C), or explicitly as lists (denoted by $[x_1, \ldots, x_n]$). The state space of the DEC-Alpha is also shown in Fig. 3. The function *reg* represents the 32 integer registers. The 32 floating point registers *freg* are just added for completeness but are not important for this paper. Note that the memory mem_α is byte-oriented and addressed by quads. The macros for *mem* define how to read and write quads into the byte oriented memory mem_α. Here, l_i denotes the i-th element of a list l (l_0 is the first element), $split_n(l)$ splits a list l into a list of lists of length n (the last list may contain less than n elements), and $concat(l)$ concatenates all lists in a list l of lists. Note that l_i is defined iff i is less than the length of l. The macros for reading and writing the byte-oriented memory specify therefore how to map the memory *mem* of the intermediate language to the memory mem_α of the target languages. It only requires an identification of sorts and operations of the static part of the intermediate language specification with those used by the machine language specification. E.g. the sorts *INT* and *ADDR* of the intermediate language are both identified with *QUAD*. The operations $\oplus_I$ and $\oplus_A$ used in the specification for the intermediate language are for the purpose of the paper additions on integers and addresses. They are both mapped to $\oplus_Q$-operation on quads. Furthermore, the program is in the memory. Therefore, the program counter *pc* contains the address of the next instruction to be executed.

Fig. 4 shows some state transitions of the DEC-Alpha Machine. The macros *rb* and *rc* are defined analogously as *ra* using functions *regb* and *regc* accessing the other registers encoded in an arithmetic instruction. The second instruction shows an addition where the second operand is a 16-bit constant directly encoded in the instruction. LDQ loads a register *ra* using address register *rb* and relative address *disp*. LDA works similarly but does not access the memory and loads the content of *rb* plus the constant *disp*. The constant may be shifted to the higher

[3] In particular, this might also indicate whether the second operand is a constant directly encoded in the instruction.

State Space

$reg:$	$RADDR \to QUAD$	
$freg:$	$RADDR \to QUAD$	
$mem_\alpha:$	$QUAD \to BYTE$	
$pc:$	$QUAD$	
Auxiliary Functions:		
$opcode:$	$LONG \to OPCODE$	opcode
$rega:$	$LONG \to ?RADDR$	register a
$type:$	$LONG \to ?TFCODE$	operation type
$immed:$	$LONG \to ?WORD$	constant operand
$disp:$	$LONG \to ?WORD$	const. rel. address
$imbyte:$	$LONG \to ?BYTE$	byte in ZAP-instr.

Macros

$$mem(a) \triangleq concat([mem_\alpha(a), \ldots, mem_\alpha(a \oplus_Q 7)])$$
$$instr_\alpha(a) \triangleq concat([mem_\alpha(a), \ldots, mem_\alpha(a \oplus_Q 4)])$$
$$mem(a) := q \triangleq \; mem_\alpha(a) := split_8(q)_0$$
$$\vdots$$
$$mem_\alpha(a \oplus_Q 7) := split_8(q)_7$$
$$pc \textbf{ is } \mathsf{ADD} \triangleq opcode(instr(pc)) = 010000$$
$$pc \textbf{ is } \mathsf{ADDQ} \triangleq pc \textbf{ is } \mathsf{ADD} \wedge type(instr(pc)) = \mathsf{0x020}$$
$$Proceed \triangleq pc := pc \oplus_Q 4$$
$$ra \triangleq rega(instr(pc))$$

Fig. 3. State Space of the DEC-Alpha Processor Family and Some Macros

```
if $pc$ is ADDQ then
    $reg(rc) := reg(ra) \oplus_Q reg(rb)$
    $Proceed$
endif
if $pc$ is ADDI then
    $reg(rc) := reg(ra) \oplus_Q SExt_{16}(immed(pc))$
    $Proceed$
endif
if $pc$ is LDQ then
    $reg(ra) := mem(reg(rb) \oplus SExt_{16}(disp))$
    $Proceed$
endif
if $pc$ is LDA then
    $reg(ra) := reg(rb) \oplus SExt_{16}(disp)$
    $Proceed$
endif
if $pc$ is LDAH then
    $reg(ra) := reg(rb) \oplus LogShift(SExt_{16}(disp), 16))$
    $Proceed$
endif
```

```
if $pc$ is STQ then
    $mem(reg(rb) \oplus SExt_{16}(disp)) := reg(ra)$
    $Proceed$
endif
if $pc$ is ZAP then
    $reg(rc) := zerobytes(reg(ra), imbyte(pc))$
    $Proceed$
endif
if $pc$ is SLL then
    $reg(rc) := LogShift(reg(ra), immed(pc)\langle 58 : 63\rangle)$
    $Proceed$
endif
if $pc$ is BR then
    $reg(ra) := pc \oplus 4$
    $pc := pc \oplus_Q 4 \oplus_Q LogShift(SExt_{21}(disp, 2))$
endif
if $pc$ is JMP then
    $reg(ra) := pc \oplus 4$
    $pc := reg(rb) \wedge_Q$ #fffffffc
endif
```

Fig. 4. Some State Transition Rules for the DEC-Alpha

16 bits. STQ is the instruction dual to LDQ and stores ra. The ZAP instruction explicitly sets some bytes in a quad word to zero. This is modeled by the function $zerobytes : QUAD \times BYTE \to QUAD$. The i-th byte in $zerobyte(q, b)$ is the zero byte iff the i-bit of b is 1, otherwise it is equal to i-th byte of q. The need of this instruction will become clear later (cf. Section 5). The SLL-instruction is the logical left shift. The last 6 bits of the register rb determine the number of bits to be shifted[4]. The jump instruction BR is a relative jump. Its operand is a 21-bit relative jump address directly encoded in the instruction. Since the alignment of instructions, it is multiplied by 4. The address of the instruction after the jump instruction is stored in register ra. The JMP instruction also stores in register ra this address. However, the jump target is contained as absolute address in register rb. Due to alignment restrictions, the last two bits are set explicitly to 0 using the bitwise conjunction $\wedge_Q$ for quadwords.

Note that these formulations can be extended in straightforward way to the semantics processor pipelines and instruction-level parallelism.

[4] $l\langle n : m\rangle$ denotes the sublist $[l_n, \ldots, l_m]$ of list l.

$X :=_I Y$	$\rightarrow \bullet$ {$\mathsf{STQ}\ Y,(0)X$}	Rule 1
$X :=_A Y$	$\rightarrow \bullet$ {$\mathsf{STQ}\ Y,(0)X$}	Rule 2
$addr_{c16} :=_I Y$	$\rightarrow \bullet$ {$\mathsf{STQ}\ Y,(c16)R30$}	Rule 3
$addr_{c16} :=_A Y$	$\rightarrow \bullet$ {$\mathsf{STQ}\ Y,(c16)R30$}	Rule 4
int_{c16}	$\rightarrow X$ {$\mathsf{LDA}\ X,(c16)R31$}	Rule 5
int_{c32}	$\rightarrow X$ {$\mathsf{LDA}\ X,(c32.L)R31;\ \mathsf{ZAP}\ X,\#\mathrm{fc},X;\ \mathsf{LDAH}\ X,(c32.H)X$}	Rule 6
$addr_{c16}$	$\rightarrow X$ {$\mathsf{LDA}\ X,(c16)R30$}	Rule 7
$addr_{c32}$	$\rightarrow X$ {$\mathsf{LDA}\ X,(c32.L)R31;\ \mathsf{ZAP}\ X,\#\mathrm{fc},X;\ \mathsf{LDAH}\ X,(c32.H)X;\mathsf{ADDQ}\ X,R30,X$}	Rule 8
$ld_I(Y)$	$\rightarrow X$ {$\mathsf{LDQ}\ X,(0)Y$}	Rule 9
$ld_A(Y)$	$\rightarrow X$ {$\mathsf{LDQ}\ X,(0)Y$}	Rule 10
$ld_I(c16)$	$\rightarrow X$ {$\mathsf{LDQ}\ X,(c16)R31$}	Rule 11
$ld_A(c16)$	$\rightarrow X$ {$\mathsf{LDQ}\ X,(c16)R31$}	Rule 12
$X +_I Y$	$\rightarrow Z$ {$\mathsf{ADDQ}\ X,Y,Z$}	Rule 13
$X +_A Y$	$\rightarrow Z$ {$\mathsf{ADDQ}\ X,Y,Z$}	Rule 14
$X +_I int_{c16}$	$\rightarrow Z$ {$\mathsf{ADDI}\ X,\#c16,Z$}	Rule 15
$X +_A addr_{c16}$	$\rightarrow Z$ {$\mathsf{ADDI}\ X,\#c16,Z$}	Rule 16

Fig. 5. Some Transformation Rules for Code Selection as Term Rewrite Rules

3 Generation of Compiler-Back Ends

Generating binary code from intermediate languages works in two phases: First, the *code selection* replaces the instruction sequence of each basic block by a machine language instruction sequence (except jumps). Second, the *assembly phase* linearizes the basic blocks and replaces jump instructions, if needed at all. We focus here on the code selection. The assembly phase is discussed in Section 6. The basic idea of term-rewriting in code selection is to apply term-rewrite rules of the form $t \rightarrow X$ or $t \rightarrow \bullet$ and associate with each rule a target code sequence $m_1, \ldots, m_n$ to be produced if the rule is applied. The first form of the rule is applied to expressions t while the latter is applied to instructions t. The term t may contain variables which are denoted by capital letters $X, Y, \ldots$ During application of a rule, each of these variables is associated with a register containing the value of the expression that is substituted for it. One may associate costs to each rule. A dynamic programming algorithm then determines the cost-optimal rule cover. However, this is not important for the purpose of correctness of the transformation.

Example 3. Fig. 5 shows some of the transformation rules for the code selection phase from our intermediate language programs to DEC-Alpha machine code. We used symbolic machine code for denoting the DEC-Alpha machine instructions. However, if register assignment is performed during code generation, this is just an abbreviation for the binary instruction format. $c_{32}.L$ and $c_{32}.H$ denote the lower and higher 16 bits of $c32$, respectively. It should be noted that some rules are specializations of other rules, e.g. rule 3 specializes rule 1. Usually, this is reasonable if one operand is a small constant (e.g. 16-bit constant). Special attention is required to understand rules 6 and 8. One could easily accidently forget the ZAP instruction. However, the LDA instruction automatically sign-extends the loaded constant (cf. Fig. 4), i.e. if bit 15 of $c32$ is 1, then all leading bits are set to 1. Hence, the following LDAH-instruction does not work properly because it expects that bits 16-63 are all 0. This is ensured by the ZAP-instruction. Observe that such kinds of compiler bugs are hard to identify. We found such a bug in a back-end written by one of our students using the proof strategies discussed in Section 5.

$addr_{28} :=_I ld_I(addr_{28}) +_I int_1$	(11)		$X = R2$	LDQ $R2,(28)R30$
$addr_{28} :=_I R2 +_I int_1$	(15)	$X = R2$	$Z = R2$	ADDI$R2,\#1,R2$
$addr_{28} :=_I R2$	(3)	$X = R2$		STQ $R2,(28)R30$
•				

Fig. 6. Code Generation for $addr_{0x001c} :=_I ld_I(addr_{0x001c}) +_I int_{0x0001}$

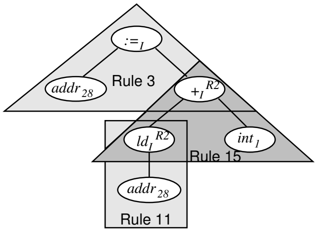


Fig. 7. Rule Cover corresponding to the Term-Rewrite of Fig 6

We show now how to apply the above rules to generate code. For each rule application a register has to be assigned to the RHS of a term-rewrite rule if it is a variable. Some of the registers are forbidden. In the DEC-Alpha processor family register $R31$ always contains 0 and cannot be written. According to conventions of the DEC-Alpha processor family the base address *base* is stored in register $R30$. Hence this register is also not assigned. Suppose we want to generate code for the instruction $addr_{0x001c} :=_I ld_I(addr_{0x001c}) +_I int_{0x0001}$. Fig. 6 shows a possible term-rewrite. The first column contains the actual term to be rewritten, the second column contains the rule that is applied, the third column shows the matching substitutions for the LHS of the rule, the fourth column shows a register assignment, and the last column shows the corresponding code generated. Reading the last column from top to bottom yields the machine instruction sequence implementing the intermediate language instruction. For any of the applied rules there are alternatives: Instead of applying rule (11) rule (7) is also applicable for loading the address constant in a register. Then, rule (10) has to be applied later. Similarly, rule (7) could be applied instead of rule (3). The final rule application would then be rule (1). Instead of applying rule (15) at the second step, one could have applied rule (5) to load the integer constant 1 to a register and then apply rule (13).

One property is that the applied rules overlap in exactly one node of the instruction tree, cf. Fig. 7 for the term-rewrite in Fig. 6. The values of these nodes must be stored in registers. A generator for code selection computes for each instruction of the intermediate program such a rule cover, assigns registers to the overlapping nodes (these are used during term-rewriting for assigning registers to variables), and plans the order of rule applications. Then, the term-rewrite is actually executed as demonstrated by Example 3. Formally the register assignment *ra* is used as a substitution of the variables in the term-rewrite rule by registers.

4 Correctness of Program Transformations

The notion of compiler or translation correctness is often defined as refinement of programming language constructs. In particular each state transition defined by a source language concept (e.g. a conditional statement) must be implemented in exactly the same way by the target machine. However, this may forbid some global or interprocedural optimizations (although for the purpose of this paper it might be sufficient). The notion of compiler correctness took in *Verifix* longer than expected. A more extensive discussion can be found in [24,21].

Observable Behaviour. From a compiler user's viewpoint, only the input/output relation of the program is of interest. Each program has such an interaction with an environment which we call *observable behaviour.* In terms of ASMs the states are projected to the I/O-relevant dynamic functions, e.g. the input/output streams (*observable states*). The *observable behaviour* of a program consists only of the observable states and state transitions between them induced by the more fine semantics. It is an abstraction of the ASM semantics as state transition system and therefore also a state transition system. Compiler users usually only require that the target program preserves the observable behaviour of the source program.

Resource Limitations. Since usually machine resources are limited while it is easy to write e.g. Java programs that would consume more than 10TByte memory, the target programs may exceptionally stop because of memory overflow. We therefore came up with the following notion of correctness: Let τ be a program of the target language with the observable behaviour $(I, Q, \rightarrow)$ and σ be a program of the source language with observable behaviour $(I', Q', \rightarrow')$. τ *preserves the observable behaviour* of σ *up to resource limitations* iff there is a relation $\phi \subseteq Q \times Q'$ such that for any finite or infinite sequence $q_0 \rightarrow q_1 \rightarrow \cdots$ of τ with $q_0 \in I, q_1, q_2, \ldots \in Q$ there is a finite or infinite sequence of states $q'_0 \rightarrow q'_1 \rightarrow \cdots$ of σ with $q'_0 \in I'$ and $q_i \phi q'_i$ for all i except possibly for the last state (if the sequence of observable states of τ is finite). This means that τ halts with violation of resource limitations. Fig. 8 visualizes this definition.

The preservation of observable behaviour up to resource limitations is transitive and therefore can be applied stepwise for the different phases in a compiler. It might be even useful for the purpose of proving correctness to introduce new intermediate languages that are not used by a compiler. If we speak in the following about source and target language, this may be one intermediate language (before the transformation) and the next intermediate language (after the transformation).

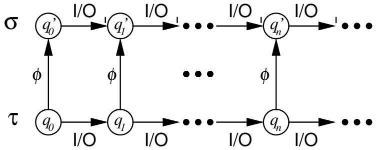


Fig. 8. Preservation of Observable Behaviour

Example 4. Consider the following language (called BBMIX) which is the union of the intermediate language in Example 1 and of basic block graphs of DEC-Alpha machine instructions (except jumps) which is obtained before assembling the binaries. The notion of programs, procedures, and basic blocks is analogous to Example 1. However, the set of instructions is the union of the set of intermediate language instructions (see Table 1) and the set of DEC-Alpha instructions except DEC-Alpha jump-instructions. Expressions are enriched by register access $rg(i)$ (abbreviated as Ri). A basic block may contain instruction sequences such as LDQ $R2, (28)R30; addr_{28} :=_I R2 +_I int_1$.

For the dynamic semantics, we extend expression evaluation by access to registers, i.e. $eval(Ri) = reg(i)$ and otherwise use all the state transition rules from the intermediate language (cf. Fig. 2) and the DEC-Alpha (except jumps, cf. Fig. 4) but with the *Proceed*-macro from the intermediate language. Note that except the expression evaluation for registers nothing need to be added for the definition of BBMIX. Anything else can be derived completely from the language definition for the intermediate language and the DEC-Alpha machine language. The intermediate language and the basic block graphs with DEC-Alpha machine instructions are now just sub-languages from BBMIX. The transformation rules in Fig. 4 are now program transformations within BBMIX. E.g. applying only Rule 11 to the instruction $addr_{28} :=_I ld_I(addr_{28}) +_I int_1$ would yield the instruction sequence LDQ $R2, (28)R30; addr_{28} :=_I R2 +_I int_1$. Therefore we can identify source and target language.

Remark 1. We were able to perform this process of language unification using ASM-semantics under rather general conditions, see [53]. It is independent on the concrete instruction sets of the intermediate language and the target language, it only requires a notion of expression in the intermediate language and the notion of registers in the target language. Note that source-to-target transformations and local optimizing transformations can be dealt within the same way.

Correctness of Program Transformations. A program transformation is *correct* iff any target program τ obtained by the program transformation from a source program σ preserves the observable behaviour of σ up to resource limitations. The correctness proof for program transformations follows the idea of simulation proofs similar to those in complexity and computability theory. One has to define a relation ρ between the states of programs of the target language and states of programs of the source language that is compatible to the relation ϕ on the observable behaviours of the target and source programs, respectively. This could be the same language as demonstrated by Example 4. The first simulation in Fig. 9 shows the conditions on ρ if the observable behaviour in the source and target program does not change (i.e. the diagrams must commute). The second simulation shows if there is exactly one observable state transition in the target and source program.

Local and Global Correctness. If a program transformation replaces a program fragment ψ' by another program fragment ψ then the state initial at ψ' and ψ and final at ψ' and ψ are in relation ρ, respectively. For a simulation proof

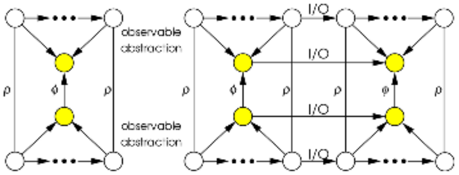
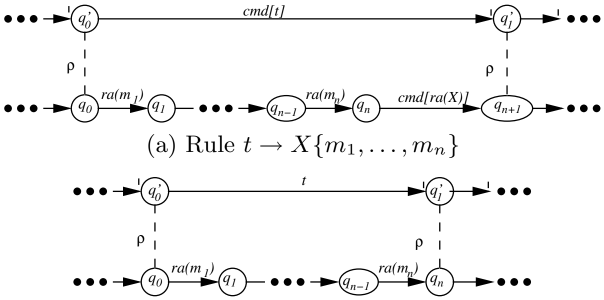


Fig. 9. Simulation Proofs



(a) Rule $t \to X\{m_1, \dots, m_n\}$

(b) Rule $t \to \bullet\{m_1, \dots, m_n\}$

Fig. 10. Local Correctness of Term-Rewrite Rules w.r.t register assignment *ra*

two properties have to be shown: First one has to show that concrete program transformations are correct in the above sense (*local correctness*). Second, the single simulations stemming from program transformations must be sequentially composable (*global correctness*). We now apply these notions for code selection. In the following we assume that the intermediate language and the target language are united in the way as demonstrated by Example 4, for details see [53]. This has the advantage we can focus on the application of a single transformation rule. The complete simulation follows by induction on the number of applied transformation rules. Since we focus on the application of a single transformation rule, we have a mapping ψ from the addresses of the instruction in the source program and the addresses in the target program. We can now define the relation ρ: Let Q' and q be the sets of states before and after the single application of this transformation rule, respectively. Two states $q \in Q$, $q \in Q'$ are said to be *corresponding* iff $[\![f]\!]_q = [\![f]\!]_{q'}$ for all dynamic functions except for *ip* and registers containing dead values w.r.t. q'[5]. The idea behind corresponding states is that their relevant memory is isomorphic. We define $q\rho q'$ iff q and q' are corresponding states and $\psi([\![ip]\!]_{q'}) = [\![ip]\!]_q$.

We first define local correctness. Let Q' and Q be defined as above, and *ra* be the register assignment computed during the planning phase of the code selection. A term-rewrite rule of the form $t \to X\{m_1, \dots, m_n\}$ is *locally correct* w.r.t. *ra* iff for all states $q_0, \dots, q_n \in Q$ such that $q_0' \models instr(ip) = cmd[t]$,

[5] A value is *dead* if it is not needed as operand by instructions executed later.

$q_0 \to q_1 \to \cdots \to q_{n-1} \to q_n$, $q_0 \models \mathit{instr}(\mathit{ip}) = \mathit{ra}(m_1), \ldots, q_{n-1} \models \mathit{instr}(\mathit{ip}) = \mathit{ra}(m_n)$, and $q_n \models \mathit{instr}(\mathit{ip}) = \mathit{cmd}[\mathit{ra}(X)]$ there is state $q'_0 \in Q'$ such that $q_0 \rho q'_0$ and the following two conditions are satisfied:

i. $[\![\mathit{eval}(t)]\!]_{q'_0} = [\![\mathit{reg}(\mathit{ra}(X))]\!]_{q_n}$ and

ii. q_n and q'_0 are corresponding states. Note that $\mathit{reg}(\mathit{ra}(X))$ must contain a dead value in q'_0.

This definition implies $q_{n+1} \rho q'_1$ since expression evaluation in the intermediate language is free of side-effects.

A term-rewrite rule $t \to \bullet\{m_1, \ldots, m_n\}$ is *locally correct* w.r.t. *ra* iff for all states $q_0, \ldots, q_n \in Q$ such that $q_0 \to q_1 \to \cdots \to q_{n-1} \to q_n$ where $q_0 \models \mathit{instr}(\mathit{ip}) = \mathit{ra}(m_1), \ldots, q_{n-1} \models \mathit{instr}(\mathit{ip}) = \mathit{ra}(m_n)$ there is a state q'_0 such that $q_0 \rho q'_0$ and the following two conditions are satisfied:

iii. q_n and q'_1 are corresponding states.

iv. $q'_0 \models \mathit{instr}(\mathit{ip}) = t$ and $[\![\mathit{instr}(\mathit{ip})]\!]_{q_n} = [\![\mathit{instr}(\mathit{ip})]\!]_{q'_1}$.

Thus it holds $q_n \rho q'_1$, cf. Fig. 10(b).

Global correctness can be proven under rather general conditions. In [53] we have proven that the following theorem holds for any transformation from basic block oriented intermediate languages with expressions to register machines:

Theorem 1 (Global Correctness of Simulation). *Let $\mathcal{T}$ be a set of term-rewrite rules specifying the transformation in a code-selection, σ be a source program annotated with register assignment ra, a rule cover, and a schedule on the order of applying rules, and τ be the target program obtained by executing the program transformations according to the schedule. If each rule in $\mathcal{T}$ is locally correct and for any applied rule $t \to X\{m_1, \ldots, m_n\}$, register $ra(X)$ (the register assigned to hold the value for t) does contain a live value, then τ preserves the observable behaviour of σ.*

The proof has two stages. First, it is by induction on the number of applied term-rewrite rules as described above. The corresponding simulation is then shown by induction on the number of executions of the program fragment the transformation was applied to (actually more recent work on infinite state sequences indicates that it is indeed co-induction, cf. [22]). The inductive step requires the local correctness conditions for term-rewrite rules.

The global correctness theorem is rather independent of the languages and only requires the notion of expressions and instruction pointers stemming from basic-block graph based intermediate languages, and the notion of register stemming from register-based target processor architectures, i.e. the same requirements as for code selection by term-rewriting. The precondition on live values can be analyzed upon compilation and is therefore part of the verification of the back-end implementation. It is a good candidate for program checking approaches [15,17]. However, proving the local correctness conditions is language-dependent and should be done for each compiler.

5 Proof Strategies for Correctness of Code Selection

Typical compiler-backends often require several thousand term-rewrite rules. The correctness of term-rewrite rules therefore cannot be proven manually. The main idea to prove mechanically correctness of term-rewrite rules is to symbolically execute the LHS of a term-rewrite rule and the code to be generated from it. For each of the two classes of term-rewrite rules there is one simple proof strategy.

Consider first term-rewrite rules of the form $t \to X\{m_1, \ldots, m_n\}$. We have basically to ensure that Condition (i) is satisfied and that no other register changes its content. The variables in the term-rewrite rule stand for registers containing some live values. We therefore use each of these variable names as symbolic register names. The expression evaluation $eval(t)$ is executed symbolically with these symbolic register names using the definition of Fig. 2 extended by reading register access as described by Example 4. Furthermore, the instruction sequence $m_1, \ldots, m_n$ is also evaluated symbolically using the state transition rules of Fig. 4. Then, the result will be checked whether $reg(X)$ contains the result of the symbolic evaluation of $eval(t)$ and nothing else (except the instruction pointer) changed. The requirement on the instruction pointer is satisfied as long as the machine instruction sequence does not contain jump instruction.

Example 5. Consider Rule 6. The symbolic evaluation of a 32-bit integer constant yields $eval(int_{c32}) = SExt_{32}(c32)$. Suppose that $c32 = [b_{31}, \ldots, b_0]$. Then $SExt_{32}(c32) = [\underbrace{b_{31}, \ldots, b_{31}}_{32 \text{ times}}, b_{31}, \ldots, b_0]$. Symbolic execution of the first instruction $\mathsf{LDA}\ X, (c32.L)R31$ yields

$$\begin{aligned} reg(X) &= reg(R31) \oplus SExt_{16}(c32.L) \\ &= 0 \oplus SExt_{16}([b_{15}, \ldots, b_0]) \\ &= [\underbrace{b_{15}, \ldots, b_{15}}_{48 \text{ times}}, b_{15}, \ldots, b_0]) \end{aligned}$$

Then, the instruction $\mathsf{ZAP}\ X, \#fc, X$ is executed:

$$\begin{aligned} reg(X) &= zerobytes(reg(X), 11111100) \\ &= zerobytes([\underbrace{b_{15}, \ldots, b_{15}}_{48 \text{ times}}, b_{15}, \ldots, b_0], 11111100) \\ &= [\underbrace{0, \ldots, 0}_{48 \text{ times}}, b_{15}, \ldots, b_0] \end{aligned}$$

Finally, $\mathsf{LDAH}\ X, (c32.H)X$ is executed:

$$\begin{aligned} reg(X) &= reg(X) \oplus LogShift(SExt_{16}(c32.H)), 16) \\ &= [\underbrace{0, \ldots, 0}_{48 \text{ times}}, b_{15}, \ldots, b_0] \oplus LogShift([\underbrace{b_{31}, \ldots, b_{31}}_{48 \text{ times}}, b_{31}, \ldots, b_{16}], 16) \\ &= [\underbrace{0, \ldots, 0}_{48 \text{ times}}, b_{15}, \ldots, b_0] \oplus [\underbrace{b_{31}, \ldots, b_{31}}_{32 \text{ times}}, b_{31}, \ldots, b_{16}, \underbrace{0, \ldots, 0}_{16 \text{ times}}] \\ &= [\underbrace{b_{31}, \ldots, b_{31}}_{32 \text{ times}}, b_{31}, \ldots, b_0] \end{aligned}$$

All other dynamic functions except *ip* do not change, because the transition rules of ASMs explicitely specify state changes. In particular each dynamic function that is not updated remains unchanged.

This proof strategy always works for this kind of transformation rules since in a term-rewrite rule $t \to X\{m_1, \ldots, m_n\}$ the term t is just an expression and therefore does not cause side-effects.

Now, we consider term-rewrite rules of the form $t \to \bullet\{m_1, \ldots, m_n\}$. Here, a similar strategy as above is used. The difference is that the execution of t causes state changes (in particular in the memory) and these state changes must be the same as caused by the machine instruction sequence $m_1, \ldots, m_n$. Therefore, we symbolically execute the state transition caused by t using the state transition rules of Fig. 2 and symbolically execute the machine instruction sequence using the state transition rules of Fig. 4 and the fact $\mathit{base} = \mathit{reg}(R30)$, i.e., the base address is stored in register $R30$. We use symbolic register addresses and compare the resulting state changes according to condition (iii).

Example 6. Consider Rule 3. According to the state transitions in Fig. 2, after execution of $\mathit{addr}_{c16} := Y$, we obtain the only state change

$$\begin{aligned}\mathit{mem}(\mathit{eval}(\mathit{addr}_{c16})) := \mathit{eval}(\mathit{rg}(Y)) &= \mathit{mem}(\mathit{base} \oplus \mathit{SExt}_{16}(c16)) := \mathit{reg}(Y)\\ &= \mathit{mem}(\mathit{reg}(R30) \oplus \mathit{SExt}_{16}(c16)) := \mathit{reg}(Y)\end{aligned}$$

According to the state transitions of Fig. 4, executing $\mathsf{STQ}\ Y, (c16)R30$ would yield the state transition: $\mathit{mem}(\mathit{reg}(R30) \oplus \mathit{SExt}_{16}(c16)) := \mathit{reg}(Y)$ As one can see, these are exactly the same updates which are executed.

If more than one machine instruction is executed then the updates stemming from the machine instructions are composed. Again, this proof strategy always works for term-rewrite rules of the form $t \to \bullet\{m_1, \ldots, m_n\}$ because here t is an instruction and it is the state transitions that are of interest.

These two proof strategies can easily be implemented. The implementation can be parameterized with the semantics of the intermediate and the target language. The size of the equalities to be proven could explode exponentially in n, the number of machine-instructions to be generated. However, this number is usually rather small.

6 Assembly

For a complete back-end remains to discuss assembly. The binary has to be mapped into the linear memory of the target machine. Thus, each label of a basic block is mapped to the address of the first instruction of the basic block and each jump-instruction has as an argument the address of the basic block associated with the jump target. There are three possible cases to map jumps: First, two basic blocks bb_1 and bb_2 are mapped consecutively and bb_2 is the single successor of bb_1. Then, no jump is necessary. If the jump instruction and the jump target are close enough, a relative jump can be made, i.e. the relative address of the jump target is directly encoded in the jump instruction. Otherwise, the jump target has to be loaded as a constant into a register. The registers used for loading this constant must not contain live values. Note that all registers assigned by local register assignment in the code selection phase satisfy this property. Hence registers with these properties can be determined at compile time.

$jmp(L) \to \bullet\ \{\}$	if $l = current \oplus_Q 4$	Rule 17
$jmp(L) \to \bullet\ \{\mathsf{BR}\ Ri, \#l_{41} \cdots l_{61}\}$	if $-2^{22} \le l < 2^{22}$	Rule 18
$jmp(L) \to \bullet\ \{\mathsf{LDA}\ Ri, (l.LL)R31$ $\mathsf{ZAP}\ Ri, \#fc, Ri$ $\mathsf{LDAH}\ Ri, (l.LH)Ri$ $\mathsf{BR}\ Rj, \#000001$ $\mathsf{ADD}\ Ri, Rj, Ri$ $\mathsf{ADD}\ Ri, \#08, Ri$ $\mathsf{JMP}\ Rj, Ri\}$	if $-2^{31} \le l < -2^{22}$ or $2^{22} \le l < 2^{31}$	Rule 19
$jmp(L) \to \bullet\ \{\mathsf{LDA}\ Ri, (l.HH)R31$ $\mathsf{SLL}\ Ri, \#10, Ri$ $\mathsf{LDA}\ Ri, (l.HL)Ri$ $\mathsf{SLL}\ Ri, \#10, Ri$ $\mathsf{LDA}\ Ri, (l.LH)Ri$ $\mathsf{SLL}\ Ri, \#10, Ri$ $\mathsf{LDA}\ Ri, (l.LL)Ri$ $\mathsf{BR}\ Rj, \#000001$ $\mathsf{ADD}\ Ri, Rj, Ri$ $\mathsf{ADD}\ Ri, \#08, Ri$ $\mathsf{JMP}\ Rj, Ri\}$	if $l < -2^{31}$ or $l \ge 2^{31}$	Rule 20

where $current = addrbb(labjmp(L)) \oplus_Q lenbb(lab(jmp(L)))$, $l = addrbb(L) - current$, $l.LL$, $l.LH$, $l.HL$, and $l.HH$ are the 1st, 2nd, 3rd, and 4th sixteen bits of l, and Ri. Rj are registers which don't contain live values

Fig. 11. Mapping of Jumps

Example 7. We have to map basic block graphs consisting of DEC-Alpha machine instructions and symbolic jumps to a linearized assembly program. The mapping $addrbb : LABEL \to QUAD$ maps each label of a basic block to a relative address. The mapping $lenbb : LABEL \to QUAD$ maps each basic block (identified by its unique symbolic label) to its length in the binary code. These mappings have to be computed by a compiler. Note that this implies that the address of the jump instruction *current* and the relative distance l to the jump target can be computed at compile time[6]. The transformations in Fig. 11 are used for mapping unconditional jumps. The transformations for conditional jumps are analogous. The first transformation is used if the successor block is mapped consecutively, the second is used if the relative distance to the jump target can be encoded as a 21-bit relative address. The last two transformations load the jump target directly into register Ri. Note that the registers required for this transformation must be free. The instruction $\mathsf{BR}\ Rj, \#000001$ is required to load $pc \oplus_Q 4$ into register R_j. The two ADD-instructions adjust the correct address of the jump target since it must be relative to the address of the last instruction of the basic block. Note that this address is given by *current*. It is not difficult to prove that these transformations are locally correct. The proofs follow the same strategy as in Fig. 10(b).

Similar to the code selection phase, the compiler performs a planning phase where it computes *addrbb*, for each basic block the transformations applied to its jump instructions, and a function $jumps : LABEL \to QUAD$ that computes

[6] $lab(jmp(L))$ is the label of the basic block that contains the jump instruction $jmp(L)$ where the transformation rule is applied.

the size of these jump instructions. For the example of the DEC-Alpha processor, it holds:

$$\mathit{jumps}(L) = \begin{cases} -4 & \text{if Rule 17 is applied to the jump in } \mathit{lab}^{-1}(L) \\ 0 & \text{if Rule 18 is applied to the jump in } \mathit{lab}^{-1}(L) \\ 24 & \text{if Rule 19 is applied to the jump in } \mathit{lab}^{-1}(L) \\ 40 & \text{if Rule 20 is applied to the jump in } \mathit{lab}^{-1}(L) \end{cases}$$

Note that $\mathit{lenbb}(L) = \mathit{length}(\mathit{lab}^{-1}(L)) \otimes_Q 4 \oplus \mathit{jumps}(L)$ since each instruction requires 4 bytes space (*length* denotes the length of a sequence). The total size of the code is $\mathit{lenbb}(L_0) + \cdots + \mathit{lenbb}(L_m)$ where $L_0, \ldots, L_n$ are the labels of all basic blocks in the program. The assembly phase might optimize this total code size. Alternatively, other criteria might be used.

For the correctness of the assembly phase, it is not hard to see that it is sufficient to prove the conditions in Fig. 12. The first condition states that instructions of a basic block are mapped consecutively without gaps in the same order as in the basic block. The second condition states that two basic blocks do not overlap. The last condition states that the transformation rules of Fig. 11 are correctly applied and that $\mathit{jumps}(L)$ is large enough to store the jump instructions of the basic block with label L. Note that all conditions of the transformation rules can be checked at compile time. Together with the local correctness of the transformations for the jumps this implies by induction on the number of state transitions the global correctness of the assembly phase.

The technique of program checking may be used for checking the conditions in Fig. 12. Note that checking these proof obligations except the last one is independent of the concrete target language because the functions instr_α addrbb, *lenbb*, *length*, and *jumps* must be computed for any register based target language and basic-block based intermediate language. Checking the last condition requires only the knowledge of the transformation rule to be applied and the size of an instruction (which is 4 in the example of DEC-Alpha). It is not hard to see that this checking algorithm requires time $O(n + k \log k)$ where n is the number of instructions in the target program and k is the number of basic blocks. Checking the first and the third condition in Fig. 12 requirerconstant time (for a single instruction), hence they require a total of time $O(n)$. For checking the second condition, the labels are sorted according to their addresses. Then, it is sufficient to check the second condition only for the consequetive labels.

7 Related Work

The kind of code generation discussed in Section 3 was first introduced as bottom-up rewrite systems (BURS) [40]. Several works improve this technique. Emmelmann implemented this technique in BEG [14,13] and adds algebraic identities and uses dynamic programming to find cost-optimal rule covers. [44] uses tree automata to execute the term-rewriting. This has the additional advantage that completeness of the specification w.r.t. the intermediate language can be

$\mathit{instr}(L,i) = \mathit{instr}(\mathit{addrbb}(L) + i \cdot 4)$ for all labels L, $0 \leq i < \mathit{length}(\mathit{lab}^{-1}(L))$ and $\mathit{instr}(L,i)$ is not a jump instruction

$\mathit{addrbb}(L_1) - \mathit{addrbb}(L_2) < \mathit{lenbb}(L_2)$ for all labels L_1, L_2, $L_1 \neq L_2$

For each transformation rule $\mathit{JMP}(L) \to \bullet\{m_1, \ldots, m_n\}$ if cond applied in a basic block with label L'

$$\mathit{jumps}(l) = (n-1) \cdot 4 \wedge \mathit{cond} \wedge \mathit{instr}(\mathit{addrbb}(L') + l \cdot 4) = m_1 \wedge \wedge \cdots \wedge \mathit{instr}(\mathit{addrbb}(L') \oplus_Q (l+n-1) \cdot 4) = m_n$$
$$\text{where } l = \mathit{length}(\mathit{lab}^{-1}(L')) - 1$$

Fig. 12. Proof Obligations for Correctness of Assembly Phase

decided efficiently. Nymeyer et. al. [38] use A^*-search in order to find rule covers. None of these works discussed the correctness of code generation.

Correctness of compilers was first considered in [31]. They discussed the compilation of arithmetic expressions. Samet used an approach that we now call translation validation [45,46,48,47,49]. There are a number of works using denotational semantics, e.g. [8,33,39,43,52]. Other works use the approach of refining language constructs, e. g. [6,7,9,27,32,34,50], or structural operational semantics, e. g. [11]. Often these works consider single phases of a compiler. E.g. [50] discusses intermediate language generation. The code selection phase for generating binary code is often not considered in works on compiler correctness. [32] discusses the compilation of a stack-based intermediate language into a register based assembly language. [34,27] consider the Transputer as target-machines. Each of these works check transformations in hand-written compiler back-ends. They don't discuss correctness of transformation rules used by generated term-rewrite based compiler back-ends.

Program checking in code generators was independently developed from us in the area of safety-critical systems [41] and is called *translation validation* by them. The difference to general compilers is that their target code has a special form and it mainly consists of an implementation of a finite state machine. Zuck et. al. extended the ideas to validate certain optimizing transformations [54,56,55,2,23]. Necula uses a similar approach for checking local optimizations in basic blocks [36]. Glesner and Blech apply translation validation to constant-folding [18]. Glesner et. al. use translation validation for correctness for the lexical analysis of the GNU C compiler [19] and generalize the approach of [15] to code generators for embedded systems [20]. However, translation validation does not help to identify erroneous transformation rules - it just tells that something in the compilation went wrong.

In contrast to our work, works on verifying compiler optimization focuses on specific parts of a compiler. Blech and Glesner prove the correctness of some optimizations using Isabelle/HOL [4,3]. Lacey et. al. use temporal logic for this purpose [28,29]. These works have in common that they do not change the language level. Strecker shows the correctness of transformations from a subset of Java (μJava) to Java Byte Code using Isabelle/HOL [51]. Schmid et. al. showed the proof for Java except threads [50] but used a paper and pencil approach. Blech, Glesner et.al. show the correctness of translations of SSA-intermediate languages to machine languages[5]. Poetzsch-Heffter and Gawkowski propose a similar approach for C[42]. Both approaches use Isabelle/HOL. They assume that there is one-to-one correspondence between machine operations and intermediate language operations. Dold, v. Henke and Goerigk developed in *Verifix* a completely verified compiler (in binary code) for a LISP subset[12]. The proofs were checked using PVS. Leinenbach, Paul, and Petrova developed in the framework of the *VeriSoft*-project a verified macro-expansion based compiler for a Pascal/C-subset using Isabelle/HOL[30]. A final remark: Proof Carrying Code [35] and Certifying Compilers [37,10] check necessary conditions for the correctness of compilation while *Verifix* and translation validation approaches check sufficient conditions.

8 Conclusions

We have shown how the transformations in compiler back-ends can be completely verified. It requires abstract state machine specifications for the semantics of the intermediate and target languages, respectively. The local correctness of the transformation rules can then be mechanically verified by using two proof strategies that suffice if the transformations for code selection and assembly are given as term-rewrite rules. The composition as simulation proofs is guaranteed under rather general requirements to intermediate and target languages, respectively. However, for the code selection the compiler has to check whether no register is written that contains a live value, i.e., a value that is still required. Having annotations to the intermediate program giving a rule cover for each instruction, a register assignment w.r.t. a rule cover, and a schedule specifying the order of application of term-rewrite rules, this can easily be checked independently of the compiler. Similarly, the compiler has to check the conditions in Fig. 12 for the assembly phase. This suggests to use techniques of program checking for that purpose. If this program checker and the module actually performing the term-rewriting is verified, it is guaranteed that any code generated by a compiler actually preserves the observable behaviour of the intermediate program.

It should be noted that the proof strategies for term-rewrite rules also work for local optimizations. However, for global optimizations such e.g. code motion, it does not work since they cannot be expressed as local transformation rules on trees. Instead, graph-rewrite rules should be used. Adding edges stemming from dataflow information might result in local graph transformations as e.g. be used in optimizer generators [1]. The same idea could be applied when instruction-scheduling techniques are applied. For pipelined architecturs and instruction-level parallelism SSA-graphs are the more suitable intermediate representation. First steps towards this direction are made[5]. However, it does not yet cover the full power of instruction scheduling approaches. This is subject to further research.

The strength of formal approaches was demonstrated by finding a serious bug in the specification of DEC-Alpha back-end developed by a student project. It was an erroneous version of the transformation for loading large constants discussed in Section 5. It was accidently overseen that loading 16-bit constants into register automatically sign-extends the constant. If correctness proofs fail, error messages should provide claims that have to be proven for ensuring local correctness of transformation rules. In the above case, this would ideally produce an error message requiring to prove that the bit 15 of a constant must be 0. This is a precise hint on what went wrong. From a practical viewpoint the use of formal methods in *Verifix* turned out to be successfull.

The lessons we learned about the use of the formal methods is that they should satisfy several requirements to be successfull in practice:

- A formal method shouldn't restrict the problem to be solved in any way. E.g. in *Verifix* we ruled out denotational semantics for language semantics since it seems that there are some requirements on compositionality.
- A formal method should take into account practical requirements. E.g. the notion of correctness in *Verifix* had to take into account resource limitations because of practical needs. This notion deviates from that in more theoretical approaches.
- A formal method should not restrict in any way design decisions for systems to be build. Otherwise, it won't be accepted by practicioners. E.g., in *Verifix* we stressed that by keeping the well-established architecture of compilers.
- Tool support is necessary because of the size and complexity of the proofs to be performed. Their underlying formal method should support tools to produce helpful error messages. In *Verifix* we used PVS for that purpose.

In particular the last issue is important for an increasing acceptance of formal methods by practicioners.

Acknowledgements. I thank the two anonymous referees for their helpful comments. I'm grateful to all colleagues of the Verifix project in Karlsruhe, Kiel, and Ulm for the inspiring and fruitful discussions.

References

1. U. Assmann. Graph rewrite systems for program optimization. *ACM Transactions on Programming Languages and Systems*, 22(4):583–637, 2000.
2. C. Barrett, B. Goldberg, and L. Zuck. Run-time validation of speculative optimizations using CVC. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.
3. J. O. Blech, L. Gesellensetter, and S. Glesner. Formal verification of dead code elimination in Isabelle/HOL. In *3rd IEEE International Conference on Software Engineering and Formal Methods*. IEEE Computer Society Press, 2005. to appear.
4. J. O. Blech and S. Glesner. A formal correctness proof for code generation from SSA form in Isabelle/HOL. In *Informatik 2004*, number P-51 in Lecture Notes in Informatics, pages 449–458. Springer, 2004. Proceedings der 3. Arbeitstagung Programmiersprachen (ATPS) auf der 34. Jahrestagung der Gesellschaft fr Informatik.
5. J. O. Blech, S. Glesner, J. Leitner, and S. Mlling. Optimizing code generation from SSA form: A comparison between two formal correctness proofs in Isabelle/HOL. *Electronic Notes in Theoretical Computer Science*, to appear, 2005. Proceedings of the 4th COCV-Workshop (Compiler Optimization meets Compiler Verification).
6. E. Brger and I. Durdanovic. Correctness of compiling occam to transputer. *The Computer Journal*, 39(1):52–92, 1996.

7. E. Brger, I. Durdanovic, and D. Rosenzweig. Occam: Specification and Compiler Correctness.Part I: The Primary Model. In U. Montanari and E.-R. Olderog, editors, *Proc. Procomet'94 (IFIP TC2 Working Conference on Programming Concepts, Methods and Calculi)*. North-Holland, 1994.
8. D. F. Brown, H. Moura, and D. A. Watt. Actress: an action semantics directed compiler generator. In *Compiler Compilers 92*, volume 641 of *Lecture Notes in Computer Science*, 1992.
9. B. Buth, K.-H. Buth, M. Fränzle, B. v. Karger, Y. Lakhneche, H. Langmaack, and M. Müller-Olm. Provably correct compiler development and implementation. In U. Kastens and P. Pfahler, editors, *Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*. Springer, 1992.
10. C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 95–107. ACM Press, 2000.
11. S. Diehl. *Semantics-Directed Generation of Compilers and Abstract Machines.* PhD thesis, Universität Saarbrücken, 1996.
12. A. Dold, F. W. von Henke, and W. Goerigk. A completely verified realistic bootstrap compiler. *International Journal on Foundations of Computer Science*, 14(4):659–680, 2003.
13. H. Emmelmann. *Codeselektion mit regulär gesteuerter Termersetzung.* PhD thesis, Universität Karlsruhe, Fakultät für Informatik, GMD-Bericht 241, Oldenbourg-Verlag, 1994.
14. H. Emmelmann, F.-W. Schröer, and R. Landwehr. BEG – a Generator for Efficient Back Ends. In *Proceedings of the Sigplan '89 Conference on Programming Language Design and Implementation*, June 1989.
15. T. Gaul, A. Heberle, W. Zimmermann, and W. Goerigk. Construction of verified software systems with program-checking: An application to compiler back-ends. In A. Pnueli and Paolo Traverso, editors, *Proceedings of RTRV '99: Workshop on Runtime Result Verification*, Trento, Italy, 1999.
16. T.S. Gaul. An Abstract State Machine Specification of the DEC-Alpha Processor Family. Verifix Working Paper [Verifix/UKA/4], University of Karlsruhe, 1995.
17. S. Glesner. Using program checking to ensure correctness of compiler implementations. *Journal of Universal Computer Science*, 9(3):191–222, 2003. Special Issue on Compiler Optimization meets Compiler Verification.
18. S. Glesner and J.-O. Blech. Classifying and formally verifying. In *2nd Workshop on Compiler Optimization meets Compiler Verification COCV2003*, volume 82 of *Electronic Notes in Theoretical Computer Science*, 2003.
19. S. Glesner, S. Forster, and M. Jäger. A program result checker for the lexical analysis of the gnu c compiler. In *3rd International Workshop on Compiler Optimization meets Compiler Verification COCV2004*, Electronic Notes in Theoretical Computer Science, 2004.
20. S. Glesner, R. Geiß, and B. Bösler. Verified code generation for embedded systems. In *1st Workshop on Compiler Optimization meets Compiler Verification COCV2002*, volume 65 of *Electronic Notes in Theoretical Computer Science*, 2002.
21. S. Glesner, G. Goos, and W. Zimmermann. Verifix: Konstruktion und Architektur verifizierender bersetzer. *IT – Information Technology*, 46(5):265–276, 2004.
22. S. Glesner and W. Zimmermann. Structural Simulation Proofs based on ASMs even for Non-Terminating Programs. In *Proceedings of the ASM-Workshop, Eight International Conference on Computer Aided Systems Theory EUROCAST 2001*, Feb 2001.

23. B. Goldberg, L. Zuck, and C. Barrett. Practical issues in translation validation for optimizing compilers. *Electronic Notes in Theoretical Computer Science*, 132(1), 2005.
24. G. Goos and W. Zimmermann. Verification of compilers. In B. Steffen E.-R. Olderog, editor, *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 201–230. Springer, 1999.
25. Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Brger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
26. Y. Gurevich. May 1997 Draft of the ASM Guide. Technical Report CSE-TR-336-97, University of Michigan EECS Department, 1997.
27. C.A.R. Hoare, He Jifeng, and A. Sampaio. Normal Form Approach to Compiler Design. *Acta Informatica*, 30:701–739, 1993.
28. D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proc. 29th ACM Symposium on Principles of Programming Languages*, pages 283–294. Association of Computing Machinery, 2002.
29. D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen. Compiler optimization correctness by temporal logic. *Higher Order and Symbolic Computation*, 17(3):173–206, 2004.
30. D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *3rd IEEE International Conference on Software Engineering and Formal Methods*. IEEE Computer Society Press, 2005. to appear.
31. J. McCarthy and J.A. Painter. Correctness of a compiler for arithmetical expressions. In J.T. Schwartz, editor, *Proceedings of a Symposium in Applied Mathematics, 19, Mathematical Aspects of Computer Science*. American Mathematical Society, 1967.
32. J S. Moore. *Piton: A Mechanically Verified Assembly-Level Language.* Kluwer Academic Press, Dordrecht, The Netherlands, 1996.
33. P. D. Mosses. Abstract semantic algebras. In D. Bjørner, editor, *Formal description of programming concepts II*, pages 63–88. IFIP IC-2 Working Conference, North Holland, 1982.
34. M. Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *Lecture Notes in Computer Science.* Springer, Berlin, Heidelberg, New York, 1997.
35. G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119. ACM Press, 1997.
36. G. C. Necula. Translation validation for an optimizing compiler. In *PLDI'00: SIGPLAN Conference on Programming Language Design and Implementation*, pages 83–95. ACM, 2000.
37. G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, 1998.
38. A. Nymeyer, J.-P. Katoen, Y. Westra, and H. Ablas. Codegeneration = A^*+BURS. In *Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, pages 160–176. Springer, 1996.
39. J. Palsberg. An automatically generated and provably correct compiler for a subset of Ada. In *IEEE International Conference on Computer Languages*, 1992.

40. E. Pelegr-Llopart and S.L.Graham:. Optimal code generation for expression trees: An application of BURS theory. In *Principle of Programming Languages POPL'88*, pages 294–308. ACM, 1988.
41. A. Pnueli, O. Shtrichman, and M. Siegel. Translation validation for synchronous languages. *Lecture Notes in Computer Science*, 1443, 1998.
42. A. Poetzsch-Heffter and M. Gawkowski. Towards proof generating compilers. *Electronic Notes in Theoretical Computer Science*, 132(1), 2005.
43. W. Polak. Compiler specification and verification. In J. Hartmanis G. Goos, editor, *Lecture Notes in Computer Science*, volume 124 of *Lecture Notes in Computer Science*. Springer, 1981.
44. T. A. Proebsting. BURS automata generation. *ACM Transactions on Programming Languages and Systems*, 17(3):461–486, 1995.
45. H. Samet. *Automatically proving the correctness of translations involving optimized code.* PhD thesis, 1975.
46. H. Samet. Compiler testing via symbolic interpretation. In *ACM 76: Proceedings of the annual conference*, pages 492–497, New York, NY, USA, 1976. ACM Press.
47. H. Samet. A machine description facility for compiler testing. *IEEE Transactions on Software Engineering*, 3(5):343–351, 1977.
48. H. Samet. A normal form for compiler testing. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 155–162, 1977.
49. H. Samet. Proving the correctness of heuristically optimized code. *Communications of the ACM*, 21(7):570–582, 1978.
50. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine.* Springer, 2001.
51. M. Strecker. Formal verification of a Java compiler in Isabelle. In *Proc. Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77. Springer Verlag, 2002.
52. M. Wand. A semantic prototyping system. *SIGPLAN Notices*, 19(6):213–221, June 1984. SIGPLAN 84 Symposium On Compiler Construction.
53. W. Zimmermann and T. Gaul. On the Construction of Correct Compiler Back-Ends: An ASM-Approach. *Journal of Universal Computer Science*, 3(5):504–567, 1997.
54. L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A Translation Validator for Optimizing Compilers. In J. Knoop and W. Zimmermann, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, 2002.
55. L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. Voc: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.
56. L. Zuck, A. Pnueli, Y. Fang, B. Goldberg, and Y. Hu. Translation and run-time validation of optimized code. *Electronic Notes in Theoretical Computer Science*, 70(4), 2002.

Accurate Theorem Proving for Program Verification*

Byron Cook[1], Daniel Kroening[2], and Natasha Sharygina[3]

[1] Microsoft Research
[2] ETH Zurich
[3] University of Lugano

Abstract. Symbolic software verification engines such as SLAM and ESC/JAVA often use automatic theorem provers to implement forms of symbolic simulation. The theorem provers that are used, such as SIMPLIFY, usually combine decision procedures for the theories of uninterpreted functions, linear arithmetic, and sometimes bit vectors using techniques proposed by Nelson-Oppen or Shostak. Programming language constructs such as pointers, structures and unions are not directly supported by the provers, and are often encoded imprecisely using axioms and uninterpreted functions.

In this paper we describe a more direct and accurate approach towards providing symbolic infrastructure for program verification engines. We propose the use of a theorem prover called COGENT, which provides better accuracy for ANSI-C expressions with the possibility of nested logic quantifiers. The prover's implementation is based on a machine-level interpretation of expressions into propositional logic. COGENT's translation allows the program verification tools to better reason about finite machine-level variables, bit operations, structures, unions, references, pointers and pointer arithmetic.

This paper also provides experimental evidence that the proposed approach is practical when applied to industrial program verification.

1 Introduction

Program verification engines, such as symbolic model checkers and advanced static checking tools, often employ automatic theorem provers for symbolic reasoning. For example, the static checkers ESC/JAVA [2] and BOOGIE [3] use the SIMPLIFY [4] theorem prover to verify user-supplied invariants. The SLAM [5,6,7,8,9,10] software model-checker uses ZAPATO [11] for symbolic simulation of C programs. The BLAST [12] and MAGIC [13] tools use SIMPLIFY for abstraction, simulation and refinement. Other examples include the InVeSt [14] tool, which uses the PVS [15] theorem prover. Further decision procedures used in program verification are CVC-LITE [16], ICS [17] and VERIFUN [18].

The majority of these theorem provers use either the Nelson-Oppen [19] or Shostak [20] combination methods. These methods combine various decision procedures to provide a rich logic for mathematical reasoning.

* This paper is an extended version of [1].

T. Margaria and B. Steffen (Eds.): ISoLA 2004, LNCS 4313, pp. 96–114, 2006.

However, the fit between the program analyzer and the theorem prover is not always ideal. The problem is that the theorem provers are typically geared towards efficiency in the mathematical theories, such as linear arithmetic over the integers. In reality, program analyzers rarely need reasoning for unbounded integers. Linearity can also be too limiting in some cases. Moreover, because linear arithmetic over the integers is not a convex theory (a restriction imposed by Nelson-Oppen and Shostak), the real numbers are often used instead. Program analyzers, however, need reasoning for the reals even less than they do for the integers.

The program analyzers must consider a number of issues that are not easily mapped into the logics supported by the theorem provers. These issues include pointers, pointer arithmetic, structures, unions, and the potential relationship between these features. Additionally, because bit vectors and arrays are not convex theories, many provers do not support them. In those that can, the link between the non-convex decision procedures can be disappointing. As an example, checking equality between a bit-vector and an integer variable is typically not supported.

When using provers such as Simplify, the program verification tools must encode the features specific to programming languages into the input logic of the theorem prover, and approximate the language semantics with axioms over the symbols used during the encoding. However, using axioms to encode the language semantics has a drawback in that they can interact badly with the heuristics that are often used by provers during axiom-instantiation in order to improve performance—at the expense of accuracy.

Another problem that occurs when using provers such as Simplify or Zapato is that, when a query is not valid, the provers do not supply concrete counterexamples. Some provers provide partial information on counterexamples. However, in program verification this information rarely leads to concrete valuations to the variables in a program, which is what a programmer most wants when a program verification tool reports a potential bug in their source code.

This paper addresses the following question: *When analyzing programs, can we abandon the Nelson-Oppen/Shostak combination framework in favor of a prover that performs a basic and precise translation of program expressions into propositional logic?*

Inspired by the success of CBMC [21] and UCLID [22], this paper describes a new theorem prover called Cogent which provides direct support for queries in the form of pure ANSI-C [23] expressions together with quantifiers. Cogent largely dispenses with the mathematical theories for unbounded integers and real numbers, and the communication between theories through equivalence relations. Instead, Cogent provides machine-level accurate reasoning for the class of expressions that occur in programs and program invariants.

Much like CBMC, the implementation of Cogent is based on a direct compilation of expressions into propositional logic. When necessary (for example, in order to handle arrays with unbounded size), Cogent uses uninterpreted functions with Ackerman's encoding. A similar approach is found in UCLID. Pointers are represented as *regions* with finite vectors and offsets.

Cogent's translation allows the program verification tools to accurately reason about arithmetic overflow, bit operations, structures, unions, pointers and pointer arithmetic. Cogent can be used for different software verification applications. As an example, when applied to software model checking it can be used within the abstraction refinement framework [24,25] for abstraction, simulation, and abstraction refinement. Cogent also produces concrete counterexamples to failed proofs.

This paper makes the following novel contributions:

- We provide details on an accurate translation from C expressions together with nested quantifiers into propositional logic. While Cogent is based on parts of the CBMC source code, this paper extends it by using nondeterminism to model architecture dependent behavior. When combined with predicate abstraction, like in SLAM, this technique guarantees that a positive verification result is valid on all standard compliant architectures.
- We demonstrate that the new approach improves the performance of software model checking. In particular, we report results of replacing Slam's theorem prover Zapato with Cogent. This allows us to speed up the verification of previously checked safety properties of Windows device drivers. The speedup is caused by the improved accuracy of Cogent. Moreover, the Cogent-based model checker allows us to verify new properties that make use of bit-level constructs. In this paper, we describe a new Windows device driver bug that was found due to Cogent's improved accuracy. The Zapato-based Slam is unable to locate this bug.
- We also report the results of experiments from queries that come from extended static checking with Boogie.

The queries from Slam and Boogie differ significantly in their characteristics, which allows us to evaluate Cogent's performance under different circumstances. Slam's queries have no quantifiers but make extensive use of structures, pointers, arithmetic and bit operations. Boogie's queries, on the other hand, have nested quantifiers and some uninterpreted functions, but do not use pointer semantics.

The remainder of this paper is organized as follows: Section 2 surveys related work; Section 3 describes the algorithm used by Cogent; Section 4 presents the results of our experiments with Cogent and Slam on benchmarks from Windows device drivers; Section 5 describes the results of Cogent when used to verify conditions generated by Boogie. Section 6 concludes the paper and Section 7 discusses future work.

2 Related Work

In this work we are following the basic proof strategy used by CBMC [21] and UCLID [22]. 1) The input logic of Cogent is translated eagerly into propositional logic. 2) The resulting propositional formula is then passed to an efficient SAT solver.

The difference between our approach and UCLID is the logic supported by the provers. UCLID does not support the low-level programming language features that Cogent does. On the other hand, Cogent does not support features such as λ-abstraction, which is supported by UCLID.

The experimental application of UCLID to software verification is limited to a restricted set of theorem proving queries from software model checking in [26]. However, neither the relative effect on accuracy nor the effect on the model checking performance was measured, as UCLID was not integrated into an abstraction refinement loop.

To the best of our knowledge (beyond the experiments in [26]), no-one has evaluated the performance of an eager and purely SAT-based theorem prover implementation in abstraction-based symbolic software model checking nor extended static checking. The use of SAT for the abstraction of ANSI-C programs was suggested in [27,28]. No comparative evaluation was done, however, and no support for quantifiers was provided.

Cogent is not unique in its support for accurate reasoning for bit-vectors. Numerous tools implement bit-vector reasoning, particularly hardware verification tools (e.g., [29,30]). Some bit-vector level decision procedures have been adapted to fit into the Nelson-Oppen/Shostak's cooperating decision procedure framework (e.g., [31]). The key difference between the bit-vector support found in Cogent and these provers is that our translation fully accounts for the semantics of the ANSI-C standard [23], using non-determinism in cases where the standard does not specify the details of the machine representation of the data types.

Some program verification tools do not use general purpose theorem provers at all. For example, PREfix [32] and ESP [33] use custom symbolic simulators in which they mix their own language semantics together with the abstractions used in order to make their verification engines scale to large programs. CMC [34] uses a similar approach. Our work is motivated by these efforts. We aim to provide accurate support for the C semantics at the same level of detail as PREfix. Note that Cogent does not provide any abstractions—we expect that the program verification tool performs the abstraction, if needed, while using Cogent for symbolic reasoning.

Cogent builds on the source code of CBMC [21]. Cogent and CBMC differ in that Cogent supports quantifiers and uses non-determinism to take architecture-dependencies into account. They also differ in their intended use: Cogent is designed to be a sound theorem prover for use in any program verification engine, whereas CBMC is a program verification engine by itself.

While not related directly, this work can contribute to the predicate abstraction refinement framework with predicates that contain quantifiers, such as described in [35]. The applications proposed by the authors (hardware and software) would benefit from the accuracy provided in Cogent.

3 Encoding into Propositional Logic

In hardware verification, the encoding of arithmetic operators such as shifting, addition, and even multiplication into propositional logic using arithmetic circuit descriptions is a standard technique. We propose using this same style of encoding in COGENT. This allows us to model artifacts such as arithmetic overflow accurately.

The goal is to implement the ANSI-C standard semantics, as described in [23]. The standard purposely does not provide precise semantics. This is to allow an efficient implementation on different architectures. As an example, the behavior in the case of arithmetic overflow on signed integer types is undefined. Thus, using a true machine-like bit-encoding would be an *under-approximation* of the behavior allowed by the standard. Potentially, this can lead to incorrect verification results, making the verification tool unsound. Therefore, the answers of the prover would be only valid for architectures that use the same bit-encoding. On other architectures, the program might execute in a different way.

In order to avert this problem, we model the architecture-dependent parts of the language semantics by introducing non-determinism into the encoding. A non-deterministic choice can be encoded in propositional logic by using free, unconstrained variables. In order to decide whether to use the non-deterministic choice or not, we add additional checks to the arithmetic operators. If an operator obtains operands for which the result is architecture dependent, the result of the operator is a non-deterministic choice.

In the context of software verification, if the prover reports that the property is verified, the property holds for any architecture compliant with the standard.

3.1 Scalar Data Types

The scalar data types are encoded using a particular bit width for each data type. This bit-width is a run-time option. The arithmetic operators (e.g., addition, multiplication, division) and the bit-wise operators are transformed into corresponding arithmetic circuits using basic gates such as AND, OR, NOT. These circuits are then transformed into propositional logic.

Optimizations for Division. While a standard arithmetic circuit for addition, subtraction, multiplication and shifting provides sufficient performance, implementing an iterative division circuit using propositional logic is prohibitively expensive. We therefore implement the division and remainder operators as follows: we use non-deterministic choice to *guess* the correct result of the division, i.e., the quotient q and the remainder r, and then add constraints that these guesses are correct. I.e., we return q, r such that $q * b + r = a$. This requires one multiplication, one addition and one equality test. Note however, that the multiplication and the addition must be forced (by adding appropriate constraints) not to overflow, or wrong results would be obtained.

Arithmetic Overflow on Unsigned Types. On unsigned integer types, the ANSI-C standard requires modular arithmetic, i.e., the result is required to be a bit-encoding of $r \bmod 2^n$, where r is the result obtained with infinite precision and n is the number of bits. Using arithmetic circuits accurately models these semantics, so no non-determinism is required.

Arithmetic Overflow on Signed Types. On signed integer types, the ANSI-C standard leaves the behavior in case of arithmetic overflow undefined. In particular, the semantics of a two's complement encoding are not guaranteed.

Formally, let $\mathit{overflow}{+}(a,b)$ denote a Boolean function that is true if and only if the sum of a and b is outside the interval given by `INT_MIN` and `INT_MAX`. Let $a \oplus b$ denote the bit-vector operator for adding a and b. Let $\perp$ denote a vector of free, new variables with the same width as the addition result. The result of a signed addition is denoted by $\mathit{op\_s}{+}$.

$$\mathit{op\_s}{+}(a,b) := \mathit{overflow}{+}(a,b) ? \perp : a \oplus b$$

Note that the case-split on the overflow is translated as part of the circuit, and thus performed dynamically by the propositional logic solver, not during translation. A similar definition is used for subtraction, multiplication, and bitwise shifting.

3.2 Structures, Unions, and Bounded Arrays

Structures and small arrays are encoded in a straight-forward manner by recursively concatenating the bit-vectors that encode their components. Large arrays are treated like arrays with unbounded size, which is described in the next section. The prover query language contains operators to extract members from a structure and to replace members. If an array index operation is out of bounds, the value of the index operator is a vector of free variables, i.e., it is non-deterministically chosen. In contrast to that, when using a conventional theorem prover such as SIMPLIFY, arrays and structures are typically encoded using uninterpreted functions and axioms. This requires expensive heuristics for quantifier instantiation.

Unions are encoded using a pair of bit-vectors. The first bit-vector is as wide as the widest bit-vector of any of the union members. It encodes the value of the union. The second bit-vector is a binary encoding of the number of the member that was used last for writing into the union. During member extraction, we check that the extracted member matches the member used for writing. If they do not match, the value of the member extraction operator is a vector of free variables.

3.3 Unbounded Arrays

Programs may allocate arrays of variable size. Encoding such an array using a bit-vector is infeasible. Thus, we model unbounded arrays as uninterpreted functions using Ackermann's reduction, as done in CBMC [21]. Note that the contents of the array are still interpreted as bit-vectors.

3.4 Pointers

We encode the value of a pointer using two bit-vectors. Let p denote a pointer type expression. The first bit-vector, denoted by $p.o$, encodes the object the pointer points to, while the second bit-vector, denoted by $p.i$, encodes an index within that object using two's complement. The width of $p.i$ is the same as the width used for the integer type. The width of $p.o$ is dynamically adjusted to accommodate the number of objects mentioned within the query. The object bit-vector consisting of all zeros is used to encode a `NULL` pointer[1].

The offset bit-vector is used to encode the position of the pointer within the object. In case of an array consisting of elements of a scalar data type, this value is equal to the array index, independent of the size of the scalar data type. Structures consisting of n fields with scalar data types are treated like an array with n elements, even if the types of the individual fields have different widths.

If arrays and structures are nested, the offset bit-vector is equal to the number of the scalar type variable inside the nested data structure.

Address Operator and Pointer Arithmetic. The encoding above models the semantics of the ANSI-C pointer operators accurately. The unary `&` operator returns the address of the object passed as operand. The operand may contain field access and array index operators. These operators are handled by adjusting the index bit-vector. As the array index may be a variable, the formula built for the index bit-vector may require addition and multiplication.

The pointer arithmetic operands only adjust the index bit-vector, never the object bit-vector. The logic includes predicates that allow checking for overflow and underflow on pointer arithmetic operations, if desired.

Function Pointers. Functions mentioned in the query are assigned object numbers just as variables. However, the ANSI-C standard provides no semantics for arithmetic on pointers pointing to functions. Thus, the pointer arithmetic operators return non-deterministic results when applied to pointers that point to functions.

Relational Operators. When checking equality between two pointers, as specified by the ANSI-C standard, the object bounds have to be considered. If the index bit-vector of the pointer is not within the object, we call the pointer out-of-bounds. As a special case, the index bit-vector of the pointer can be exactly one element beyond the end of the object. We call such a pointer an off-by-one pointer. In case of a pointer pointing to the `NULL` object, any index bit-vector other than zero is considered to be out of bounds. These comparisons are done dynamically by encoding an arithmetic circuit for the relations on the index bit-vector and the object size, which may be a variable.

We form an equation that dynamically distinguishes the following cases:

- If both pointers are within their bounds, the result of the comparison is equal to bitwise equality of both components of the pointer.

[1] Note that the ANSI-C standard prohibits dereferencing a `NULL` pointer. It is a common misunderstanding that dereferencing `NULL` will result in the value zero.

– If both pointers point to the same object (i.e., the object bit-vectors are bitwise equal) and both pointers are within their bounds or off-by-one, the result of the comparison is equal to bitwise equality of the index bit-vector.
– Otherwise, the result is a free, unconstrained variable.

When checking the other relations (greater than, and so on), the standard requires that the two pointers must point to the same object. Also, the pointer must be within the bounds or off-by-one. The result of the comparison is a non-deterministic choice if either check fails.

3.5 Quantifiers

The sections above describe the translation of formulae into propositional logic. In these formulae, all variables are assumed to be implicitly universally quantified. However, some program analysis tools make use of nested quantifiers. In most cases, Cogent is able to rewrite the input query in such a way that the quantifiers can be encoded directly into propositional logic with fresh variables and Skolemization. In the worst case, Cogent will translate the input formula into propositional logic with quantifiers (called *quantified Boolean formulae* or QBF) instead of the standard propositional logic—we believe that this case will not occur frequently in practice.

3.6 Examples

In order to summarize the techniques above, consider the following examples. Given the formula Q

```
p+x!=q || &(p->y) == &((q-x)->y)
```

let `p` and `q` be two pointers to a structure containing a member `y`. Let `x` be an integer variable.

This formula is translated into propositional logic as follows. First, two new Boolean variables α and β are allocated for the two operands of the OR operator. Then, we add the following constraint:

$$Q \iff \alpha \vee \beta$$

We then add the constraints for the left-hand side operand of the OR operator. We allocate bit-vectors for $p.i$, $p.o$, $q.i$, $q.o$, and x. We assume that n is the number of elements of simple type in the structure, and that $\otimes$ is the bit-vector multiplication operator.

$$\alpha \iff (p.i \oplus (x \otimes n) \neq q.i) \vee (p.o \neq q.o)$$

Note that this constraint does not contain the bounds check for the object pointed to by `p` and `q`.

For the encoding of the right-hand side of the OR operator, suppose that `y` is the second member of the structure. Thus, the index bit-vector is increased by one when taking the address of `p->y`.

$$\beta \iff (p.i \oplus 1 = (q.i \ominus (x \otimes n)) \oplus 1) \wedge (p.o = q.o)$$

This simple example illustrates the complexity of mixing pointer arithmetic with structures and arrays. In contrast to our tool, existing decision procedures are unable to handle even such simple examples.

In order to illustrate an invalid query generating a counterexample, consider the formula R

```
!(p==a+2 && q==a+n && p==q)
```

where `a` is an array, `p` and `q` are pointers, and `n` is an integer. Again, we first assign fresh Boolean variables α_2 β_2, and γ_2 to the operands of the AND operator:

$$R \iff !(\alpha_2 \wedge \beta_2 \wedge \gamma_2)$$

The encoding of the constraints for the pointer arithmetic is done similarily as above for α. The object `a` is assigned a number. Suppose this number is 1. When passed to a SAT solver, we obtain a satisfying assignment with `n` $= 2$, a value of 1 for the object of `p` and `q`, and a value of 2 for the offset of `p` and `q`.

4 Application to Software Model Checking

One popular approach to software model checking is called *counter-example guided abstraction refinement* (CEGAR). SLAM, for example, implements CEGAR for the C programming language. CEGAR implementations [24,25,36,37,13, 14,12] often use automatic theorem provers to implement the abstraction and refinement components of this algorithm. In this section, we briefly describe the CEGAR approach, and then present results of an experiment with SLAM where we have replaced the theorem prover ZAPATO with COGENT.

4.1 Software Model Checking with Counter-Example Guided Abstraction-Refinement

Predicate abstraction [38,39] is a method for systematically constructing conservative abstractions of software. It abstracts data by only keeping track of certain predicates on the data. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. The predicate abstraction of software is usually automated. For example, in SLAM, the predicate abstraction is implemented in a module called C2BP [40,5].

In practice, the set of predicates must be discovered by trial-and-error. Typically, CEGAR implementations guess the initial set of predicates. If the abstraction is computed using an insufficient set of predicates, then the model checker will find a false error in the abstraction—called a *spurious trace*. There are two sources of spurious traces: 1) the set of predicates is insufficient, and 2) C2BP trades accuracy for efficiency.

SLAM first uses NEWTON [6] to symbolically simulate the entire trace and determine if it is spurious. If the trace is spurious, then NEWTON searches for additional predicates which could eliminate the trace in a refined abstraction.

If no new predicates are found, SLAM concludes that the spurious trace is caused by the inexact abstraction done by C2BP. It then invokes another refinement method, called CONSTRAIN [7]. CONSTRAIN symbolically examines each step of the trace in isolation and attempts to refine the abstract transition relation in order to improve the accuracy of the abstraction using the predicates that are available.

By default, NEWTON and CONSTRAIN both use the theorem prover ZAPATO [11]. As is done in [26], C2BP does not call a theorem prover. Instead, it uses a module called FASTCOVERING that implements a form of parallel inference.

4.2 Experiments with SLAM

We have integrated COGENT with SLAM and compared the results to SLAM using its original theorem prover, ZAPATO. Note that, in our integration, C2BP still uses FASTCOVERING. FASTCOVERING is currently an extremely weak inference engine that produces poor abstractions when uncommon symbols (like the C bitwise operations) appear in the sets of predicates. As a consequence, SLAM/COGENT is at a disadvantage over SLAM/ZAPATO, as the abstraction of bitwise operations must be done in a needlessly inefficient manner with CONSTRAIN. For a more optimal result, FASTCOVERING should perform an analysis similar to COGENT in order to provide better abstractions.

4.2.1 Comparing the Model Checking Results

In order to compare the overall effect of COGENT on SLAM we ran SLAM/COGENT on 308 model checking benchmarks and compared the results to SLAM/ZAPATO. The results are given in Fig. 1.

Model checking result	SLAM/ZAPATO	SLAM/COGENT
Property passes	243	264
Time threshold exceeded	39	17
Property violations found	17	19
Cases of abstraction-refinement failure	9	8

Fig. 1. Comparison of SLAM/ZAPATO to SLAM/COGENT on 308 device driver correctness model checking benchmarks. The time threshold was set to 1200 seconds.

The SLAM/COGENT performs considerably better that SLAM/ZAPATO. Notably, the number of cases where SLAM exceeded the 1200 second time threshold was reduced by half. As a result, the reduced timeouts led to two additional bugs being found. The cases where SLAM failed to refine the abstraction (as described in detail in [7]) was effectively unchanged.

During SLAM's execution, the provers actually returned different results in some cases. This is expected, as the provers support different logics:

- ZAPATO provides support for uninterpreted functions together with UTVPI integer arithmetic [41]. In addition, ZAPATO supports expressions with pointers only through axioms and a heuristic for dynamic axiom instantiation.
- COGENT, on the other hand, supports full arithmetic over bit vectors together with a more accurate handling of pointers and structures. COGENT is strictly more accurate than ZAPATO.

For this reason, there are queries that ZAPATO can prove valid and COGENT can prove invalid (e.g., when overflow is ignored by ZAPATO), and vice-versa (e.g., when validity is dependent on pointer arithmetic or non-linear uses of multiplication). Thus, it is difficult to compare the accuracy of ZAPATO to COGENT. We have, however, compared the overall performance of the two provers and found that COGENT is usually more than 2x slower than ZAPATO. On 2000 theorem proving queries ZAPATO executed for 208s, whereas COGENT ran for 522s. We can therefore conclude that the performance improvement in Fig. 1 is indicative that, while COGENT is slower, COGENT's increased accuracy allows SLAM to do less work overall.

4.3 Checking New Properties of Windows Drivers

During the formalization of the kernel API usage properties that SLAM is used to verify [25], a large set of properties were removed or not actively pursued due to inaccuracies in SLAM's theorem prover. For this reason the results in Fig. 1 are not fully representative of the improvement in accuracy that SLAM/COGENT can give.

In order to demonstrate this improved accuracy, we developed and checked several new safety properties that would have resulted in too many false bugs being reported in SLAM/ZAPATO. Fig. 2 contains an example of such a property, written in SLAM's event-based property language called SLIC [42]. It makes use of COGENT's treatment of bit vectors, structures and pointers. This rule checks that a Windows device driver always sets a special bit in a field of a structure to `0` before returning from its `AddDevice` callback routine.

This property has the effect of instrumenting three events into the driver when SLAM performs symbolic model checking:

- Calls from the device driver to the kernel function `IoCreateDevice`, which (in the case the function returns successfully) causes an assignment of `1` to the variable `created`.
- Calls from the device driver to the kernel function `IoDeleteDevice`, which causes an assignment of `0` to the variable `created`.

```
// The variable "created" is 0 when the special variable pdevobj is not
// pointing to something that has been allocated.  It is set to
// 1 when it is.
state
{
    int created = 0;
}

// IoCreateDevice will, if successful, place the pointer pdevobj in the
// handle passed to it.
IoCreateDevice.exit
{
    if ($return==STATUS_SUCCESS) {
        created = 1;
    }
}

IoDeleteDevice.exit
{
    created = 0;
}

// If the driver has an AddDevice callback, it will be called fun_AddDevice
#ifdef fun_AddDevice
fun_AddDevice.exit
{
    // pdevobj is the pointer returned the environment model
    // for IoCreateDevice
    if (created && (pdevobj->Flags & DO_DEVICE_INITIALIZING) != 0) {
        abort "AddDevice routine failed to set ~DO_DEVICE_INITIALIZING flag";
    }
}
#endif
```

Fig. 2. Slic device driver safety property using C bit operations

– Returns from the device driver's `AddDevice` callback routine.[2] When this event occurs, a check (under the condition that the device object has been allocated) ensures that the driver has negated the `DO_DEVICE_INITIALIZING` flag in the device object structure that was allocated.

This rule is checked together with a `main` function that calls the driver's `AddDevice` routine from an unspecified state, and a set of non-deterministically abstracted models of the kernel functions that the driver might call.

Fig. 3 displays the environment model for the function `IoCreateDevice` that is used while checking device drivers with Slam. This function can return any

[2] `AddDevice` is referred to as a C macro called `fun_AddDevice` in the property. Before Slam is used to perform model checking, an initial scan of the driver's source code is done and special callbacks found during this pass are defined using the C macro language. These macros are then called from the properties and kernel environment model.

```
NTSTATUS
IoCreateDevice(
    DRIVER_OBJECT * DriverObject,
    unsigned long int DeviceExtensionSize,
    UNICODE_STRING * DeviceName,
    DEVICE_TYPE DeviceType,
    unsigned long int DeviceCharacteristics,
    unsigned int Exclusive,
    DEVICE_OBJECT **DeviceObject
    )
{
    switch (MakeNondeterministicChoice()) {
      case 0:  (*DeviceObject) = pdevobj;
               pdevobj->Flags |= DO_DEVICE_INITIALIZING;
               return STATUS_SUCCESS;
      case 1:  (*DeviceObject) = NULL;
               return STATUS_INSUFFICIENT_RESOURCES;
      case 2:  (*DeviceObject) = NULL;
               return STATUS_OBJECT_NAME_EXISTS;
      default: (*DeviceObject) = NULL;
               return STATUS_OBJECT_NAME_COLLISION;
    }
}
```

Fig. 3. Nondeterministic environment model of Windows kernel function `IoCreateDevice` for device driver verification with Slam

of four possible return values. In the case that it returns `STATUS_SUCCESS` it sets the `DO_DEVICE_INITIALIZING` flag in `pdevobj`'s `Flags` field to `1`.

We checked this new property on 15 Windows device drivers using both Slam/Zapato and Slam/Cogent. When using Zapato, Slam found false errors in each driver. When using Cogent as the prover, Slam was able to verify the correctness of all but one driver. In the case of this one driver, Slam produced a counterexample that pointed to a real and previously unseen bug.

5 Application to Extended Static Checking

Boogie [3] is an implementation of Detlef *et al.*'s notion of *extended static checking* [43] for the C# programming language. Extended static checkers attempt to automatically verify manually added pre- and post-conditions in code. It can also be used to ensure that client-code respects the pre-conditions, and does not assume too much of the post-conditions. Boogie, using a notion of *weakest-preconditions*, computes *verification conditions* that can be checked by an automatic theorem prover. Boogie uses Simplify to formally validate the conditions.

In order to demonstrate the applicability of Cogent to extended static checking, we have applied it to verification conditions generated by Boogie and compared the results to those of Simplify. The runtimes in seconds are given in Fig. 4.

Benchmark #	Cogent	Simplify
1	0.010s	0.029s
2	0.013s	0.029s
3	0.012s	0.028s
4	0.041s	0.042s
5	0.573s	0.452s
6	0.001s	0.026s
7	0.002s	0.026s
8	0.002s	0.027s
9	0.042s	0.045s
10	0.043s	0.051s
11	0.030s	0.045s
12	0.002s	0.025s
13	0.003s	0.026s
14	0.093s	0.100s
15	26.217s	15.735s
16	0.001s	0.024s
17	0.001s	0.025s
18	0.002s	0.026s
19	0.010s	0.030s
20	0.013s	0.029s
21	0.013s	0.028s
22	0.042s	0.043s
23	0.571s	0.455s
24	0.001s	0.026s
25	0.001s	0.026s
26	0.003s	0.027s
27	0.042s	0.045s
28	0.044s	0.050s
29	0.030s	0.045s
30	0.002s	0.025s
31	0.003s	0.026s
32	0.093s	0.111s
33	27.772s	16.311s
34	0.001s	0.024s
35	0.001s	0.025s
36	0.002s	0.025s
37	0.010s	0.038s
38	0.014s	0.030s
39	0.012s	0.029s
40	0.042s	0.042s
41	0.573s	0.457s
42	0.002s	0.026s
43	0.001s	0.025s
44	0.002s	0.027s
45	0.042s	0.045s
46	0.043s	0.050s
47	0.030s	0.045s
48	0.002s	0.025s
49	0.003s	0.026s
50	0.092s	0.112s
51	30.813s	70.763s
52	0.001s	0.024s
53	0.001s	0.024s
54	0.001s	0.025s

Fig. 4. Comparison of Simplify and Cogent on 54 verification conditions generated by Boogie

Unlike we did in the case of Slam, we have not yet fully integrated Boogie and Cogent. For the purpose of this experiment we first annotated the variable names in the input C# programs with their types—Boogie currently does not pass any type information down to the theorem prover. We then ran Boogie on the programs and collected the verification conditions. We converted the queries from Simplify's input format into the syntax of Cogent, and removed the axioms and artifacts of the Simplify-specific encoding. Note that the comparison is not quite fair: Simplify's execution time includes parsing, whereas parsing and translation is not included in the Cogent execution time.

The verification conditions mix both finite and infinite types together with references. Objects of unbounded types were encoded with uninterpreted functions and axioms. The verification conditions did not contain any pointer arithmetic, nor C-style unions. They did, however, contain some examples with bit-level operations. In particular, one C# program models a microprocessor (as described in some detail in [44]) and makes heavy use of bit-level programming constructs.

For the harder queries, COGENT was faster in one instance, whereas SIMPLIFY was faster in two. Unlike SLAM, BOOGIE does not use the results of the validity checks during its analysis, so the increased accuracy provided by COGENT does not improve the overall performance of BOOGIE.

Note that C# provides an unsafe extension, in which pointer arithmetic and other C-like features can be used. This is, for example, how C# calls C code. Using the increased accuracy of COGENT for low-level programming features, BOOGIE could potentially analyze mixtures of unsafe and safe code.

6 Conclusion

Automatic theorem provers are often used by program verification engines. However, the logics implemented by these theorem provers are not typically ideal for the program verification domain. In this paper, we have described a new prover that accurately supports the type of reasoning that program verification engines require.

The prover's strategy is to directly encode input queries into propositional logic. This encoding accurately supports bit operations, structures, unions, pointers and pointer arithmetic, and pays particular attention to the sometimes subtle semantics described in the ANSI-C standard. We have detailed the prover's translation of queries into propositional logic. We have also reported experimental results that demonstrate the performance and accuracy improvements of the approach. We make the tool and the bitvector benchmark files used available on the web[3] in order to allow other researchers to reproduce our results.

7 Future Work

As future work, we would like to further extend the prover with features that can be useful for symbolic program verification tools. As an example, the prover should take a query that represents a symbolic state of a program and apply a widening operation such that verification engines based on abstract interpretation [45] could potentially reach a fixpoint. Additionally, we would like to make use of interpolants [46,47] in COGENT.

A number of modern automatic theorem provers, such as CVC-LITE, ZAPATO, ICS and VERIFUN, produce proofs. These proofs can be used in cases to quickly determine *why* a query is valid. When used for symbolic simulation, this allows us to find a small set of facts that cause a trace to be spurious. SAT-solvers for propositional logic typically can produce an *unsatisfiable core* which has similar information. For this reason, COGENT is able to produce information that is similar—but not identical—to the proofs generated by traditional provers. In

[3] http://www.inf.ethz.ch/personal/kroening/cogent/

the future we would like to demonstrate that the unsatisfiable cores can provide the same benefit to symbolic simulators as the proofs.

As mentioned in Section 4, the abstraction module of Slam uses FastCovering, which is similar to [26] but optimized for speed and not precision. The motivation behind this approach is to avoid the exponential number of calls to a theorem prover—as originally proposed in [38]. As we replaced Zapato with Cogent in Section 4, we would also like to replace FastCovering with a new module that supports the same level of accuracy as Cogent.

There are a number of areas for potential performance improvement in Cogent. We would like to optimize Cogent and then perform more extensive empirical comparisons. Additionally, we would like to better integrate Cogent with Boogie and compare the approaches on a larger set of benchmarks.

Acknowledgments

The authors would like to thank Mike Barnett, Sergey Berezin, Vijay Ganesh, Rustan Leino, Madan Musuvathi, and Lintao Zhang for their ideas and comments related to this work.

References

1. Cook, B., Kroening, D., Sharygina, N.: Cogent: Accurate theorem proving for program verification. In Etessami, K., Rajamani, S.K., eds.: Proceedings of CAV 2005. Volume 3576 of Lecture Notes in Computer Science., Springer Verlag (2005)
2. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI 02: Programming Language Design and Implementation. (2002)
3. Barnett, M., DeLine, R., Fahndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology **3** (2004) 27–56
4. Detlefs, D., Nelson, G., , Saxe, J.B.: Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs (2003)
5. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: PLDI 01: Programming Language Design and Implementation, ACM (2001) 203–213
6. Ball, T., Rajamani, S.K.: Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR-TR-2002-09, Microsoft Research (2002)
7. Ball, T., Cook, B., Das, S., Rajamani, S.K.: Refining approximations in software predicate abstraction. In: TACAS 04: Tools and Algorithms for Construction and Analysis of Systems, Springer-Verlag (2004)
8. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for Boolean programs. In: SPIN 00: SPIN Workshop. LNCS 1885. Springer-Verlag (2000) 113–130
9. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: SPIN 00: SPIN Workshop. LNCS 1885. Springer-Verlag (2000) 113–130

10. Ball, T., Rajamani, S.K.: Bebop: A path-sensitive interprocedural dataflow engine. In: PASTE 01: Workshop on Program Analysis for Software Tools and Engineering, ACM (2001) 97–103
11. Ball, T., Cook, B., Lahiri, S.K., Zhang, L.: Zapato: Automatic theorem proving for predicate abstraction refinement. In: CAV 04: International Conference on Computer-Aided Verification. (2004)
12. Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread modular abstraction refinement. In: CAV 03: International Conference on Computer-Aided Verification, Springer Verlag (2003) 262–274
13. Chaki, S., Clarke, E., Groce, A., Strichman, O.: Predicate abstraction with minimum predicates. In: CHARME 03: Advanced Research Working Conference on Correct Hardware Design and Verification Methods. (2003)
14. Lakhnech, Y., Bensalem, S., Berezin, S., Owre, S.: Incremental verification by abstraction. In: TACAS 01: Tools and Algorithms for the Construction and Analysis of Systems. (2001)
15. Owre, S., Shankar, N., Rushby, J.: PVS: A prototype verification system. In: CADE 11: International Conference on Automated Deduction. (1992) Saratoga Springs, NY.
16. Barrett, C., Berezin, S.: CVC Lite: A new implementation of the cooperating validity checker. In: CAV 04: International Conference on Computer-Aided Verification. (2004)
17. Filliatre, J.C., Owre, S., Rue, H., Shankar, N.: ICS: Integrated canonizer and solver. In: CAV 01: International Conference on Computer-Aided Verification. (2001)
18. Flanagan, C., Joshi, R., Ou, X., Saxe, J.B.: Theorem proving using lazy proof explication. In: CAV 03: International Conference on Computer-Aided Verification. (2003) 355–367
19. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems **1** (1979) 245–257
20. Shostak, R.E.: Deciding combinations of theories. Journal of the ACM **31** (1984) 1–12
21. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS 04: Tools and Algorithms for the Construction and Analysis of Systems. (2004)
22. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: CAV 02: International Conference on Computer-Aided Verification. (2002)
23. International Organization for Standardization: ISO/IEC 9899:1999: Programming languages — C. International Organization for Standardization, Geneva, Switzerland (1999)
24. Kurshan, R.: Computer-Aided Verification of Coordinating Processes. Princeton University Press, Princeton (1995)
25. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In: IFM 04: Fourth International Conference on Integrated Formal Methods. (2004)
26. Lahiri, S.K., Bryant, R.E., Cook, B.: A symbolic approach to predicate abstraction. In: CAV 03: International Conference on Computer-Aided Verification. (2003) 141–153
27. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI–C programs using SAT. Technical Report CMU-CS-03-186, Carnegie Mellon University, School of Computer Science (2003)

28. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI–C programs using SAT. Formal Methods in System Design **25** (2004) 105–127
29. Aagaard, M., Jones, R., Melham, T., O'Leary, J., Seger, C.J.H.: A methodology for large scale hardware verification. In: FMCAD 02: Formal Methods In Computer-Aided Design. (2002)
30. Grundy, J.: Verified optimizations for the Intel IA-64 architecture. In: TPHOLs 00: Theorem Proving in Higher-Order Logics. (2000)
31. Barret, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for bit-vector arithmetic. In: DAC 98: Design Automation Conference. (1998)
32. Bush, W.R., Pincus, J.D., Sielaff, D.J.: A static analyzer for finding dynamic programming errors. Software—Practice and Experience **30** (2000) 775–802
33. Das, M., Lerner, S., Seigle, M.: ESP: path-sensitive program verification in polynomial time. In: PLDI 02: Programming Language Design and Implementation. (2002)
34. Musuvathi, M.S., Park, D., Chou, A., Engler, D.R., Dill, D.L.: CMC: A pragmatic approach to model checking real code. In: OSDI 02: Operating Systems Design and Implementation. (2002)
35. Lahiri, S.K., Bryant, R.E.: Constructing quantified invariants via predicate abstraction. In: Proc. of the 5th Intl. Conference on Verification, Model Checking and Abstract Interpretation (VMCAI). Number 2937 in LNCS, Springer-Verlag (2004) 267–281
36. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV 00: International Conference on Computer-Aided Verification. (2000)
37. Das, S., Dill, D.L.: Successive approximation of abstract transition relations. In: Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science. (2001) June 2001, Boston, USA.
38. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In Grumberg, O., ed.: CAV 97: Conference on Computer Aided Verification. Volume 1254 of Lecture notes in Computer Science., Springer-Verlag (1997) 72–83 June 1997, Haifa, Israel.
39. Colón, M.A., Uribe, T.E.: Generating finite-state abstractions of reactive systems using decision procedures. In: CAV 98: Conference on Computer-Aided Verification. Volume 1427 of Lecture Notes in Computer Science., Springer-Verlag (1998) 293–304
40. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian abstractions for model checking C programs. In: TACAS 01: Tools and Algorithms for Construction and Analysis of Systems. LNCS 2031, Springer-Verlag (2001) 268–283
41. Harvey, W., Stuckey, P.: A unit two variable per inequality integer constraint solver for constraint logic programming. In: Australian Computer Science Conference (Australian Computer Science Communications). (1997) 102–111
42. Ball, T., Rajamani, S.K.: SLIC: A specification language for interface checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research (2001)
43. Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended static checking. Technical Report 159, Compaq Systems Research Center (1998)
44. Gurevich, Y., Wallace, C.: Specification and verification of the Windows Card runtime environment using abstract state machines. Technical Report MSR-TR-99-07, Microsoft Research (1999)
45. Cousot, P.: Abstract interpretation. Symposium on Models of Programming Languages and Computation, ACM Computing Surveys **28** (1996) 324–328

46. Thomas A. Henzinger, Ranjit Jhala, R.M., McMillan, K.L.: Abstractions from proofs. In: POPL 04: Principles of Programming Languages, ACM Press (2004) 232–244
47. McMillan, K.: An interpolating theorem prover. In: TACAS 04: Tools and Algorithms for Construction and Analysis of Systems, Springer-Verlag (2004)

Designing Safe, Reliable Systems Using Scade

Parosh Aziz Abdulla[1], Johann Deneux[1], Gunnar Stålmarck[2],
Herman Ågren[2], and Ove Åkerlund[2]

[1] Uppsala University, department of Information Technology
box 337 SE-75105 Uppsala, Sweden
[2] Prover Technology AB, Rosenlundsgatan 54 SE-11863 Stockholm, Sweden

Abstract. As safety critical systems increase in size and complexity, the need for efficient tools to verify their reliability grows. In this paper we present a tool that helps engineers design safe and reliable systems. Systems are reliable if they keep operating safely when components fail. Our tool is at the core of the Scade Design Verifier integrated within Scade, a product developed by Esterel Technologies. Scade includes a graphical interface to build formal models in the synchronous data-flow language Lustre. Our tool automatically extends Lustre models by injecting faults, using libraries of typical failures. It allows to perform *Failure Mode and Effect Analysis*, which consists of verifying whether systems remain safe when selected components fail. The tool can also compute minimal combinations of failures breaking systems' safety, which is similar to *Fault Tree Analysis*. The paper includes successful verifications of examples from the aeronautics industry.

1 Introduction

Embedded controllers are found in an increasing number of systems. Their role consists of continuously processing flows of data coming from sensors to control various devices. The increase in size and complexity of these controllers has followed that of the systems they belong to. Manual verification is no longer an option, and non-exhaustive testing has its limits. They must be complemented with exhaustive methods, if possible in an automated way.

Formal methods such as *model checking* [10] are good candidates. They have been improving for several years within the research sector, and have recently started to reach the industry. Model checking consists of automatically verifying that a *model* representing a system meets all of its requirements. In order for the method to work, both the model and the requirements must be described formally. We present our tool, *Prover SL Data Edition* (Prover SL DE), which performs reachability analysis using SAT-based model checking [9,19]. It is integrated within several designing tools, including *Scade Suite*, a set of software tools developed by Esterel Technologies. Scade Suite includes the following tools:

- A graphical editor to build formal models and to specify safety properties. Alternatively, it is possible to translate existing models written in other languages.

T. Margaria and B. Steffen (Eds.): ISoLA 2004, LNCS 4313, pp. 115–129, 2006.

- The Scade Design Verifier, built on top of Prover SL DE, to automatically verify that models satisfy all safety properties.
- A simulation environment to interactively execute models step by step.
- A C code generator. Since the code is automatically generated from the formal model, it is correct by construction, assuming the formal model is correct and the code generator is bug-free.

Designing safe systems is important, but it is also vital to make them reliable (fault-tolerant) i.e. they must remain safe even during failures of components. The use of formal methods to prove reliability is an attractive solution, since it increases the level of confidence in the design. There are two ways to verify that systems are reliable:

- Failure Mode and Effect Analysis (FMEA). In this approach, one tries to find the consequences of failures of components. This is usually achieved by means of simulation.
- Fault Tree Analysis (FTA). This method is the opposite: one wants to find the causes of a specific safety violation. In other words, the goal is to find combinations of components which must fail in order to make the system unsafe.

Both FMEA and FTA are described in details in [22].

In this paper, we present a tool, Prover SL DE, to perform both safety and reliability analyses, using the two methods described above. We describe the process of failure injection, as well as our algorithm to perform FTA. We also demonstrate the usability of our tools on three case studies from aircraft systems, provided by our industrial partners. The analyses were performed by our partners.

Related Work. There is a number of tools to perform safety and reliability analysis of complex systems using formal methods:

FSAP/NuSMV-SA [5] uses SMV's [15] input language as the modeling language. It supports automated BDD-based [6] verification using NuSMV [8], fault tree generation and order analysis. Requirements are specified using Computation Tree Logic [3], allowing for both safety and liveness properties.

In [4], **Altarica** [13] is used to model systems. Fault trees are automatically generated and analyzed using Aralia [12], an efficient BDD-based fault tree analysis tool. Model checking is performed with Cadence Lab SMV. Safety requirement are specified in Linear Time Logic [17], a logic capable of expressing both safety and liveness properties.

Scade with Prover SL DE uses symbolic model-checking to verify safety properties. It differs from these tools in the following ways:

- We use the same formal language both for requirements and for the model, which may make it easier for system designers and safety engineers to adopt our tool.
- The SAT-solver in our tool supports rational and integer linear arithmetics, in addition to non-linear arithmetics over finite domains.

- The model-checking algorithm does not rely solely on BDDs. Although BDDs can deal with many large formulas efficiently, they are known to perform poorly in some cases. In order to be able to handle as many systems as possible, our tool uses a combination of SAT procedures [11,20] as well as BDDs.

[16] describes a methodology and a tool allowing to automatically generate fault trees from Simulink models of hardware and software systems. These fault trees must then be analyzed using a separate tool, e.g. FaultTree+. This differs from our approach since our tool computes directly minimal cut sets, whereas in [16] this task is delegated to FaultTree+ or another fault tree analysis tool.

Outline. This paper is organized as follows: We will first describe in Section 2 the modeling language used in Scade. Then, in Section 3 we will show how SAT-based model checking [9,19] is used to automatically verify that the design satisfies all requirements. Since we are also interested in designing reliable systems, we will continue in Section 4 with reliability analysis, i.e FTA and FMEA. Finally, in Section 5 we will provide examples to demonstrate the use of our tools and evaluate their performance.

2 The Modeling Language

In order to formally verify systems using model checking, one must be able to build formal models of these systems. We use Lustre [7], a synchronous dataflow language. It is a relatively simple language whose semantics are unambiguous, yet its expressiveness is sufficient for designing simple software intended for safety-critical systems, e.g. software controlling aircraft systems. Moreover, there exist translators to C [21] which are certified for use in designing safety-critical systems. Compatibility with such certified tools is a prerequisite for actual use. A *dataflow*, or *flow*, is a variable whose value can change over time. All flows are synchronized, meaning that there is a single global clock controlling when flows change. The amount of real time passing between two clock ticks is not necessarily constant. Each flow is typed: it can be Boolean, integer or real. *Nodes* combine flows to generate new flows. Several basic nodes are provided: Logic operators (AND, OR, NOT...), integer and real arithmetic operators (addition, multiplication, division...). The third type consists of timed operators:

- Delays: The `PRE` operator makes it possible to refer to the previous value of a flow. It can, for example, be used to memorize values: `A = PRE A`. The *current* value of `A` is defined to be the *previous* of A.
- Initial value: The `->` operator is used to specify the value of a flow during the first time step. Consider the following example: `A = True ->!PRE(A)`. This defines flow `A` to be initially True. After that, the value is inverted (`!` is the logical NOT operator) every time step, thus modeling a square clock signal.

A system is modeled as a node, possibly composed of several sub-nodes. Recursion is not allowed, meaning that a node may not include itself as one of its sub-nodes, or in one of its respective sub-nodes. Therefore, it is possible to "flatten" the top node by substituting their contents to sub-nodes.

Scade provides a graphical interface to create, edit and visualize nodes. There are two ways to visualize nodes: The *network view* (Figure 1) and the *state machine view*. A textual equivalent representation of Figure 1 in Lustre can be seen in figure 2. This fictitious example is a controller for the doors of a lift. Requests to open the door are received from other parts of the system. These requests are granted provided that the lift is not in motion, and that it is at level with the floor. If the open request is granted, the door is kept open until the safety conditions are violated, or a close request is received.

Lustre is also used for expressing safety requirements. The system being in a safe state is denoted by a specific Boolean flow in the model being true. The

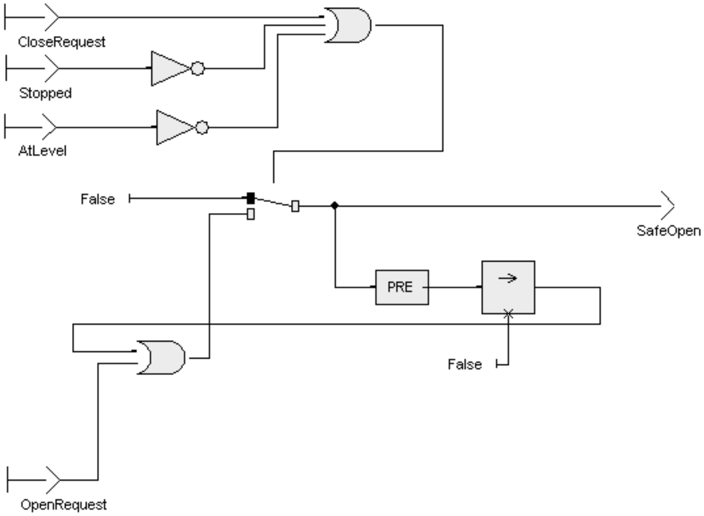


Fig. 1. Graphical representation of a lift door controller

```
node LiftDoor(OpenRequest: bool; CloseRequest: bool;
              Stopped: bool; AtLevel: bool)
returns (SafeOpen: bool) ;
let
  SafeOpen = if (CloseRequest or !Stopped or !AtLevel)
    then False
    else (False -> (pre SafeOpen)) or OpenRequest ;
tel ;
```

Fig. 2. Textual representation of the lift door controller

model checker verifies whether this flow is always true. In other words, it performs safety analysis by proving that the system constantly remains in a safe state. Popular alternatives for specifying requirements are temporal logics such as Linear Time Logic or Computation Tree Logic. See for example [14] or [15]. Our decision to use Lustre has the advantage that users need not learn an additional requirement language. Although this implies that we are limited to verifying safety properties, we consider this to be an acceptable restriction since they constitute the majority of properties used in safety and reliability analysis.

Back to our lift door example, two requirements could be:

```
OpensWhenSafe = (OpenRequest and AtLevel and Stopped) implies
                 SafeOpen;

ClosesWhenUnsafe = (!AtLevel or !Stopped) implies !SafeOpen;
```

The first requirement ensures that users do not get trapped inside the lift, that the door opens when requested if it can be done safely. The second requirement makes sure that the lift cannot harm its passengers by opening while in motion, or when not at level with the floor.

Lustre supports *assertions* to restrict the possible values of input flows. Similarly to requirements, assertions are represented by Boolean expressions which must always be True. They differ from requirements in the sense that they express assumptions about the environment of the system, which is not part of the model. In the lift door example, we may assume that the environment will never request to open and close the door simultaneously:

```
assert !(OpenRequest and CloseRequest);
```

When generating C code from a Lustre model, assertions can be translated into C assert macro calls. Assertions are also used by Scade Design Verifier to speed up the verification. Instead of verifying the model for all possible combinations of inputs, the verification is limited to those inputs satisfying the assertions.

In the next section, we describe some of the techniques used in Prover SL DE, upon which Scade Design Verifier is built.

3 Verifying Safety

Prover SL DE verifies safety properties of transition systems. We will first define the terms *transition systems* and *safety properties*, then explain how our tool performs this kind of verification.

3.1 Transition Systems

A *transition system* is a tuple (S, S_0, T), where

- S is a set of states,
- $S_0 \subseteq S$ is the set of *initial states*
- $T \subseteq S \times S$ is the transition relation.

A *safety property* P is a set of states denoting the good states.

Let $\mathit{Reach}_T(S)$ be the set of states reachable from S using the transition relation T.

We want to decide if a transition system is safe, i.e. given a transition system $M = (S, S_0, T)$ and a safety property P, is it the case that $\mathit{Reach}_T(S_0) \subseteq P$?

Lustre models are transition systems. The state of a Lustre model is denoted by the current values of all its flows. The set of initial states is specified in the model using initial value (`->`) operators. The transition relation is specified using delay operators (PRE). The set of states is the set of all assignments to flows in the model. This set is potentially infinite, because of the use of unbounded types (integers and reals). Although Lustre can express complex arithmetic expressions, Prover SL DE is limited to:

- Linear arithmetics over the set $\mathbb{Q}$ of rational numbers, i.e. expressions of the form:
  $$a_0 * C_0 + \ldots + a_n * C_n \diamond C$$
  where $a_0, \ldots, a_n$ are variables, $C, C_0, \ldots, C_n$ are constants and $\diamond \in \{=, \neq, >, <, \leq, \geq\}$.
- Non-linear arithmetics over finite domains.

When presented with a non-linear system, a safety engineer has the following options:

- Approximate the non-linear expressions by linear ones. For instance, one can replace the expression $sin(x)$ by a constraint specifying the range of $sin(x)$, i.e. $[-1, 1]$. However, this approach can cause verification to find false positives, or even worse, to miss unsafe behaviors.
- When dealing with arithmetics on integers (a common case), one can fix the size of operands. This solution is safe, provided the size is equal to he size in the implementation. However, this can significantly slow down verification.

3.2 SAT-Based Model Checking

Building an explicit representation of the reachable state space is in general not feasible (in our case it is even impossible). Instead, we represent sets symbolically using predicates. Checking for the non-reachability of a set of bad states can now be done by checking non-satisfiability of Boolean and linear arithmetic formulas. This technique is *SAT-based model checking* extended to arithmetics.

For a set of states S, let $S(s)$ be a predicate such that $s \in S \iff S(s)$.

For a sequence of states $s_0 \ldots s_n$, $\mathit{path}(s_0 \ldots s_n)$ is a predicate denoting that the sequence corresponds to a path through the graph of the transition relation.

$$\mathit{path}(s_0 \ldots s_n) = \forall i \in \{0, \ldots, n-1\} : T(s_i, s_{i+1})$$

The reachability problem for a transition system (S, S_0, T) and a safety property P can be reformulated as follows:

$$\forall n \geq 0 : \forall s_0 \ldots s_n : \mathit{path}(s_0 \ldots s_n) \wedge S_0(s_0) \Rightarrow P(s_n)$$

Two methods to solve this problem are *Bounded Model Checking* [9] and *Induction Over Time* [19].

The first method is suitable for debugging, i.e. finding errors in unsafe systems.

$$\mathit{bmc}_n(s_0 \dots s_n) = \mathit{path}(s_0 \dots s_n) \wedge S_0(s_0) \Rightarrow P(s_n)$$

It proceeds iteratively by increasing n until $\mathit{bmc}_n(s_0 \dots s_n)$ is falsifiable, in which case we have found a shortest path to a bad state. However, this method will never terminate for safe systems.

In the second method, we try to prove by induction over k that the system is safe:

- **Base case:** $\mathit{bmc}_n(s_0 \dots s_n)$ is a tautology.
- **Induction hypothesis:** $\mathit{ih}_n(s_k \dots s_{k+n}) = \mathit{path}(s_k \dots s_{k+n}) \wedge \forall i \in \{0, \dots, n\}$: $P(s_{k+i})$
- **Induction step:** $\mathit{is}_n(s_{k+n}) = \forall s_{k+n+1} : T(s_{k+n}, s_{k+n+1}) \Rightarrow P(s_{k+n+1})$

Concretely, we increase n, starting from 0, until:

$$(\forall s_0 \dots s_n : \mathit{bmc}_n(s_0 \dots s_n)) \wedge (\forall s_k \dots s_{k+n} : \mathit{ih}_n(s_k \dots s_{k+n}) \Rightarrow \mathit{is}_n(s_{k+n}))$$

If we succeed, we have proved that the system is safe. If the system is not safe, then the Bounded Model Checking step of the base case will detect it. This procedure is still incomplete: Consider the case where an unreachable loop leads to a bad state, shown in figure 3. The induction step will never succeed, even though the system is correct. This is solved [19] by modifying the *path* predicate to loop-free paths:

$$\mathit{path}(s_0 \dots s_n) = \forall i \in \{0, \dots, n-1\} : T(s_i, s_{i+1}) \wedge \forall j \in \{0, \dots, n\} : i \neq j \Rightarrow s_i \neq s_j$$

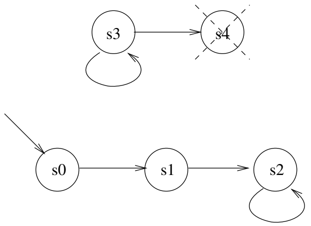


Fig. 3. A good unreachable loop leading to a bad state. The safety property includes s0, s1, s2 and s3, but not s4. s0 is the initial state.

3.3 Deciding the Satisfiability of Formulas

We have now described how to transform the problem of deciding whether a transition system is safe into deciding whether a formula is satisfiable or not. The kind of formulas we have to deal with are *math-formulas* [2]. A math-formula combines Boolean propositions and linear arithmetic predicates:

- A constant c in $\mathbb{Q}$ is a *math-term.*
- A variable v over $\mathbb{Q}$ is also a math-term.
- $c.v$ is a math-term.
- If t_1 and t_2 are math-terms, then so are $t_1 + t_2$ and $t_1 - t_2$.
- A Boolean proposition is a math-formula.
- If t_1 and t_2 are math-terms, then $t_1 \diamond t_2$ where $\diamond \in \{=, \neq, >, <, \leq, \geq\}$, is a math-formula.
- If ϕ_1 and ϕ_2 are math-formulas, then $\neg\phi_1$ and $\phi_1 \Box \phi_2$, where $\Box \in \{$*and, or, implies, not*$\}$, are math-formulas.

A naive procedure for deciding the satisfiability of a math-formula, which is a NP-hard problem [2], is to examine all satisfying assignments to the boolean variables in the formula, and for each of these solve the resulting system of linear constraints. The proof engine implements an efficient solver [1] for MATH-SAT which combines SAT techniques, such as Stålmarck's saturation method [20], Davis-Putnam-Loveland-Logemann [11], Reduced Ordered Binary Decision Diagrams [6], linear programming techniques and constraint propagation. In practice the proof engine can decide a strict superset of math-formulas, mainly due to the constraint propagation. Even if a given satisfiable formula contains non-linear predicates, the proof engine often manages to decide it. In the case of integers restricted to finite ranges, Prover SL DE converts them to bit vectors, and uses binary arithmetics to perform all operations. This method is able to handle non-linear arithmetic over finite ranges.

4 Reliability Analysis

In this section we explain shortly FTA and FMEA, and describe how to use Scade to design reliable systems.

4.1 FTA and FMEA

A failure is the inability of a piece of equipment to perform its task. Here we make a distinction between system-level and component-level failures. We restrict the use of the term "failure" to component-level failures. When the system itself fails to meet its expected performance, we say it is "unsafe", or that it violates a "safety requirement". A system is reliable when it can sustain several failures before becoming unsafe. More precisely, it is N-fault-tolerant if it remains safe unless more than N failures happen. Two popular methods to assess reliability are Failure Mode and Effect Analysis (FMEA) and Fault Tree Analysis (FTA) [22]. The term "failure mode" refers to the way a component fails. For instance, a valve may fail in different ways: It can be stuck in the opened position, in the closed, or in some intermediate position. Each way of failing is called a failure mode.

The first method, FMEA, consists of investigating the effects of failure modes. Designers specify a list of components that fail in addition to the way they fail, then the system is simulated to check if it becomes unsafe. The second method,

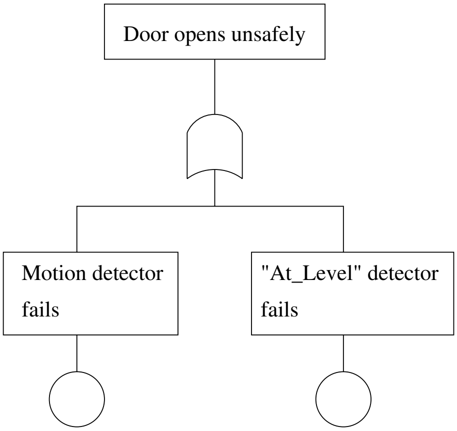


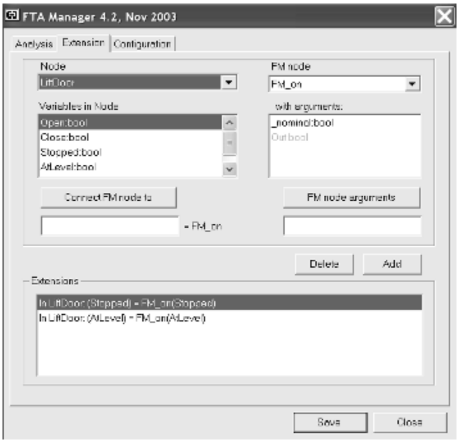
Fig. 4. A simple fault tree

FTA, can be seen as the opposite approach. It aims at finding the causes of safety violations. A fault tree (Figure 4) is a graph relating failures of components and safety violations. The root of the tree is called *Top Level Event*, and represents an event that should not occur in a safe system. In this example, the top event consists of the opening of the doors of a lift while it is moving or when it is not at the level of the floor. The leaves are called *basic events*. They represent failures of components as well as their failure mode. Here, the left basic event represents the event that the motion detector fails to report movement. The right event denotes the failure of the sensor to detect that the lift is not at the level of the floor. The internal nodes are Boolean connectives. The connective represented in this example is an OR gate. The fault tree is in fact a graphical representation of a Boolean formula satisfied when the system is unsafe. The variables in the formula denote failures of components. The goal of FTA is to find the minimal combinations of basic events leading to the top event. In other words, one wants to compute the minimal cut sets or prime implicants of the Boolean expression represented by the fault tree.

We have extended Prover SL DE to support these two methods. We will now describe how we perform reliability analysis using Scade and Prover SL DE.

4.2 Fault Injection

In order to assess the reliability of a system, its model must include failure modes. The process of adding failure modes into an existing model is called *fault injection*. We have implemented a graphical user interface (Figure 5) allowing designers to select the components susceptible to failure as well as their failure mode. This results in a new model including failure modes, which is then analyzed using the methods described in section 3. Failures of components are modeled by modifying flows representing components outputs. The original



Node: LiftDoor

FM node: FM_on

Variables in Node:
Open:bool
Close:bool
Stopped:bool
AtLevel:bool

with arguments:
_nominal:bool
Out:bool

= FM_on

Extensions:
In LiftDoor:(Stopped) = FM_on(Stopped)
In LiftDoor:(AtLevel) = FM_on(AtLevel)

Fig. 5. The fault injection panel

flows, called *nominal* flows, are replaced by modified flows, called *extended* flows. The value of an extended flow is decided by the failure mode. All possible failure modes affecting the nominal flow are modeled by a Lustre node called *failure mode node*. A typical failure mode node has two or more inputs and one output. One of the inputs is the nominal flow, and the output is the extended flow. The remaining inputs are Boolean flows called *failure mode variables* controlling which failure mode is triggered. Figure 6 represents two failure modes affecting a Boolean flow: The value of the nominal flow (`in`) is ignored and the extended flow (`out`) is set to False or True. This failure mode node can be used to model two failure modes of a switch, for instance:

- `FM_OFF`, in which case the switch acts as if it was stuck in the OFF position, or
- `FM_ON`, the switch behaves as if it was stuck in the ON position, possibly because of a short-circuit.

Note that `FM_OFF` and `FM_ON` are Boolean signals, whose values can change. This allows to model transient errors, e.g. glitches from sensors.

The result of the fault injection into the lift door model (Figure 2) is shown in Figure 7. The `FTA_` prefix marks extended flows added during fault injection. `FM_Fails_ON` is a failure mode node where a signal remains constantly True. Flows with names starting with `FM_` trigger failure modes.

4.3 FMEA in Scade

Using the same graphical interface shown in Figure 5, designers constrain the occurrence of failures. Typical kinds of constraints include:

- At most N failure modes can occur. This is equivalent to “At most N failure mode variables can switch from False to True”.

- At most N failure modes can happen simultaneously, which is the same as "At most N failure mode variables can be True at any point in time"
- Once a component fails, it never recovers and continues to fail indefinitely
- A failure mode X cannot happen.
- When failure mode X is triggered, it continues to happen for (at least, exactly, at most) T time steps.

These constraints are specified in Lustre, in a manner similar to requirements. A constraint node has a single Boolean output flow, and any number of input flows of any type. These input flows can take any value, as long as the constraint node's output remains True. Scade Design Verifier verifies that the safety requirement is always respected, assuming all constraints are met. If this is not the case, a sequence causing the system to become unsafe is returned.

FMEA can be used to gain more information about a particular combination of failures. In the example of the lift door controller, it is possible to check if the system may become unsafe when the `Stopped` sensor has glitches, i.e. the sensor provides the wrong information, but only for short amounts of time. The

```
node FM_Fails_ON_or_OFF(in: bool; FM_ON: bool; FM_OFF: bool) returns
  (out: bool)
let
   out = if (FM_ON) then True else
         if (FM_OFF) then False else
         in;
   assert not (FM_ON and FM_OFF);
tel;
```

Fig. 6. A failure mode node

```
node FTA_LiftDoor(OpenRequest : bool ; CloseRequest : bool ;
                  Stopped : bool ; AtLevel : bool,
                  FM_Stopped_Fails_ON : bool;
                  FM_AtLevel_Fails_ON : bool)
returns (SafeOpen : bool) ;
var
  FTA_Stopped: bool;
  FTA_AtLevel: bool;
let
  FTA_Stopped = FM_Fails_ON(Stopped, FM_Stopped_Fails_ON);
  FTA_AtLevel = FM_Fails_ON(AtLevel, FM_AtLevel_Fails_ON);

  SafeOpen = if (CloseRequest or !FTA_Stopped or !FTA_AtLevel)
    then False
    else (False -> (pre SafeOpen)) or OpenRequest ;
tel ;
```

Fig. 7. The model of a lift door after fault injection

safety engineer would constrain `FM_Stopped_Fails_ON` to remain True e.g. for at most three consecutive time steps, and verify if the safety requirement can be violated.

4.4 FTA in Scade

The goal of FTA is to compute the minimal combinations of failures (also called minimal cut set) causing a safety violation. Our tool proceeds by checking whether the system is safe assuming that N failure modes occur, starting with $N = 0$, and then increasing N. At each step, Scade Design Verifier verifies if the system is safe. If it is not, the Design Verifier generated a counter-example containing the values of each flow at each time step until the safety requirement was violated. From this counter-example, the set of flows representing failure modes that were triggered is extracted. These flows constitute a cut set. The operation is repeated until all cut sets smaller than a user-fixed limit have been found.

The first step, when $N = 0$, amounts to verifying that the system is safe. If it is not, then it is obviously not reliable. Otherwise N is increased to 1 and the system's safety is checked again, assuming one failure mode occurs. If the system is not safe, a counter example is generated. Since the verification was restricted to the case where one failure mode occurs, one of the failure mode variables in the counter-example must be True at some point in time. This failure mode variable represents one of the minimal cut sets of size 1. The tool continues by doing another analysis with N unchanged until no more cut sets of size 1 can be found. N is then increased, and the same steps are taken until N reaches a user-fixed limit, usually 4 or 5. The process is summarized below:

```
ComputeMCS($M$: system model, $req$: safety requirement, $N_{max}$: integer):
Let $N$ be an integer
Let $S$ be a set of cut sets
$N := 0$
$S := \{\}$
Repeat
        Let $C_1$ be the constraint:
            at most N failure mode variables become True
        Let $C_2$ be the constraint:
            no combination of failure modes found in S is triggered
        Let $c_x$ be a counter-example
        $c_x := Verify(C_1 \wedge C_2, M, req)$
        If $c_x$ is not empty (i.e. the system is not safe)
            Extract a cut set $s$ from the counter-example $c_x$
            $S := S \cup \{s\}$
        Else (i.e. the system is safe) $N := N + 1$
Until $N = N_{max}$
```

Verify($C_1 \wedge C_2, M,$ *req*) is a call to the model checker. The verification is constrained to those executions satisfying C_1 and C_2. If the system is not safe, a counter-example is returned and stored in c_x.

5 Applications

In order to evaluate the tool, our industrial partners provided a number of examples which they analyzed using our tools. We describe three of them in this section: *air inlet control*, *nose wheel steering* and *hydraulic system*. All models are designed and analyzed on widely available laptops equipped with Intel Pentium3 processors with 512MB of RAM.

5.1 Air Inlet Control

This system is a controller to automatically manoeuvre opening and closing of doors of an aircraft to regulate the inflow of air to an auxiliary power unit. Since faulty cooling of the auxiliary power unit is a hazardous event the automated manoeuvring is safety critical.

This model consists of a state transition diagram, regulating the doors movement. The system contains 21 Boolean inputs, 12 Boolean outputs and 2 rational inputs. 20 flows among the inputs are affected by fault injection, resulting in 40 new Boolean inputs. Arithmetic expressions found in this model are limited to simple comparisons.

In this case many variables represent input coming from sensors telling if doors are closed or open, or information about motor status.

One safety requirement concerns the movement of doors when landing. Landing is detected by a sensor recognizing if there is any weight on the wheels. The corresponding input flow in the model is named ”weight_on_wheel”. When this variable changes from False to True, i.e. a landing event was detected, the airflow doors must be open.

The verification, taking less than a minute, concludes that the system is safe, i.e. the requirement is respected when no components fail. It is however not reliable, since 5 different single failures and 3 double failures can make the system unsafe. This result was expected.

5.2 Nose Wheel Steering

This example is a control system to ensure suitable manoeuvrability for different aircraft operations whilst on the ground. It was originally designed in Mathworks, Matlab/Simulink, then automatically translated using tools from the Scade suite.

The Scade model includes 36 inputs (33 Boolean and 3 rational). The requirements concerns the validity of the value of the steering angle, computed by the controller. It must remain within predefined bounds. All 33 Boolean inputs are affected by failures, thus doubling the number of variables in the system after fault injection.

This requirement is fulfilled when no failures are allowed, i.e. the design is safe. This model is not expected to be reliable. Indeed, 32 minimal cut sets of size 1 were found. The analysis took about 10 minutes.

5.3 Hydraulic System

This system controls the hydraulic power supply to devices ensuring aircraft control in flight, landing gear, braking system, etc. Three independent hydraulic subsystems are shared between consuming devices in order to achieve fault tolerance. The hazardous event we want to investigate in this case is the total loss of hydraulic power.

This system was originally modeled in Altarica, whose semantics are close to Lustre's, making it easily translatable to a Scade model. However, the translation was only partly done automatically, manual intervention was still needed to complete the translation to Lustre. Since fault injection was performed on the original Altarica model, it was not performed again on the Scade model. Unlike the other examples presented in this section, the original model already takes into account failures of components. It is supposed to be 2-faults tolerant.

This analysis, which took about 3 minutes, found no single or double cut sets, 11 cut sets of size 3 and 24 cut sets of size 4.

6 Conclusion

In this paper we have presented a methodology to perform FMEA and FTA using Scade Suite and Scade Design Verifier from Esterel Technologies. Scade Design Verifier is based on the proof engine Prover SL Data Edition from Prover Technology, which has also been presented.

Future Work. Our users remarked that sequences showing violations of requirements are too complex. They contain too many variables, and it is hard to find which ones are "interesting", i.e. which variables have a key role in the unreliability of a system. This problem and several solutions are discussed in [18]. Our implementation of Fault Tree Analysis, which repeatedly calls the model checker, is currently quite naive. We plan to optimize the model checker for this kind of usage, thus possibly reducing the number of calls and hopefully speeding up each verification. Finally, we will also extend the tool to support order analysis [5].

References

1. Gunnar Andersson, Per Bjesse, Byron Cook, and Ziyad Hanna. A proof engine approach to solving combinational design automation problems. In *Proceedings of the 39th conference on Design automation*, pages 725–730. ACM Press, 2002.
2. Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In *Proceedings of the 18th International Conference on Automated Deduction*, pages 195–210. Springer-Verlag, 2002.
3. M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.
4. Pierre Bieber, Charles Castel, and Christel Seguin. Combination of fault tree analysis and model-checking for safety assessment of complex system. In *Proceedings of the fourth European Dependable Computing Conference (EDCC-4), Toulouse.* Springer Verlag, October 2002.

5. Marco Bozzano and Adolfo Villafiorita. Improving system reliability via model checking: The FSAP/NuSMV-SA safety analysis platform. In *Proceedings of the 22nd International Conference on Computer Safety, Reliability and Security [SAFECOMP 2003]*, September 2003.
6. R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, Aug. 1986.
7. P. Caspi, D. Pilaud, N. Halbwachs, and J.Plaice. Lustre: a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages, Munchen*, January 1987.
8. Alessandro Cimatti et al. NuSMV2: an opensource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer-Verlag, July 27–31 2002.
9. Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
10. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specification. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, April 1986.
11. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
12. Yves Dutuit and Antoine Rauzy. Exact and truncated computations of prime implicants of coherent and non-coherent fault trees within Aralia. *Reliability Engineering and System Safety*, 1997.
13. Alain Griffault, Sylvain Lajeunesse, Gérald Point, Antoine Rauzy, Jean Pierre Signoret, and Philippe Thomas. The AltaRica language. In *Proceedings of the International Conference on Safety and Reliability, ESREL'98*. Balkema Publishers, June 20-24 1998.
14. G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, SE-23(5):279–295, May 1997.
15. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
16. Y. Papadopoulos and M. Maruhn. Model-based synthesis of fault trees from Matlab-Simulink models. In *Proc. International Conference on Dependable Systems and Networks, 2001*, pages 77–82, 2001.
17. A. Pnueli. The temporal logic of programs. In *Proc.* 18^{th} *Annual Symp. Foundations of Computer Science*, pages 46–57. IEEE, 31 October–2 November 1977.
18. K. Ravi and F. Somenzi. Minimal assignments for bounded model checking. In *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference*. Springer-Verlag Heidelberg, April 2004.
19. M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Int. Conf. on Formal Methods in Computer-Aided Design*, volume 1954, 2000.
20. Mary Sheeran and Gunnar Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. In G. Gopalakrishnan and P. Windley, editors, *Proceedings 2nd Intl. Conf. on Formal Methods in Computer-Aided Design, FMCAD'98, Palo Alto, CA, USA, 4–6 Nov 1998*, volume 1522, pages 82–99, Berlin, 1998. Springer-Verlag.
21. Esterel Technologies. Scade suite do-178b qualified code generator. `http://www.esterel-technologies.com/products/scade-suite/do-178b-code-generation.html`.
22. W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook.* U. S. Nuclear Regulatory Commission, NUREG-0492, Washington DC, 1981.

Decreasing Maintenance Costs by Introducing Formal Analysis of Real-Time Behavior in Industrial Settings

Anders Wall[1], Johan Andersson[2], and Christer Norström[2]

[1] ABB AB, Corporate Research, Forskargränd, Västerås, Sweden
anders.wall@se.abb.com
[2] Department of Computer Science and Engineering, Mälardalen University,
Box 883, Västerås, Sweden
{johan.x.andersson, christer.norstrom}@mdh.se

Abstract. A common problem with long-lived large industrial software systems such as telecom and industrial automation systems is the increasing complexity and the lack of formal models enabling efficient analyses of critical properties. New features are added or changed during the system life cycle and it becomes harder and harder to predict the impact of maintenance operations such as adding new features or fixing bugs.

We present a framework for introducing analyzability in a late phase of the system's life cycle. The framework is based on the general idea of introducing a probabilistic formal model that is analyzable with respect to the system properties in focus, timing and usage of logical resources. The analyses are based on simulations. Traditional analysis method falls short due to a too limited modelling language or problems to scale up to real industrial systems.

This method can be used for predicting the impact caused by e.g. adding a new feature or other changes to the system. This enables the system developers to identify potential problems with their design at an early stage and thus decreasing the maintenance cost.

The framework primarily targets large industrial real-time systems, but it is applicable on a wide range of software system where complexity is an issue. This paper presents the general ideas of the framework, how to construct, validate, and use this type of models, and how the industry can benefit from this. The paper also present a set of tools developed to support the framework and our experiences from deploying parts of the framework at a company.

1 Introduction

Large industrial software systems evolve as new features are being added. This is necessary for the companies in order to be competitive. However, this evolution typically causes the software architecture to degrade, leading to increased maintenance costs. Such systems, e.g. process control systems, industrial robot control systems, automotive systems and telecommunication systems, have typically been in operation for quite many years and have evolved considerably since

T. Margaria and B. Steffen (Eds.): ISoLA 2004, LNCS 4313, pp. 130–145, 2006.

their first release and are today maintained by a staff where most of the people were not involved in the initial design of the system. These systems typically lack a formal model enabling analysis of different system properties.

The architectural degradation is a result of maintenance operations (e.g. new features and bug fixes) performed in a less than optimal manner due to e.g. time pressure, wrong competence, or insufficient documentation. As a result of these maintenance operations, not only the size but also the complexity of the system increases as new dependencies are introduced and architectural guidelines are broken. Eventually it becomes hard to predict the impact a certain maintenance operation will have on the system's behaviour. This low system understandability forces the developers to rely on extensive testing, which is time consuming, costly and usually misses a lot of bugs. The software systems we are studying have real-time requirements, which mean that it is of vital importance that the system is analyzable with respect to timing related properties, such as response times. However, bugs related to concurrency and timing are hard to find by testing [1], as they are hard to reproduce.

By introducing analyzability with respect to properties of interest, the understandability of the system can be increased. If it is possible to predict the impact of a maintenance operation, it can help system architects making the right design decisions. This leads to a decreased maintenance cost and the life cycle of the system is lengthened.

The work presented in this paper focuses on a model-based approach for increasing the understandability and analysability with respect to timing and utilization of logical resources. A model is constructed, describing the timing and behaviour of the system, based on the source code, documentation and statistical information measured on the system. This model can be used for *impact analysis*, i.e. predicting the impact a change will have on the runtime behaviour of the system. We refer to the general method as the ART Framework. This implementation of the framework is based on the ART-ML modelling language [2]. An ART-ML model is intended to be analysed using simulation. An initial case-study in presented in [3], where we used the modelling language and a simulator is used in order to analyse timing properties of an industrial system. The case-study showed the feasiblity of using the modelling language for this purpose, but also showed that analysing the simulation output required tool support. We therefore proposed the Probabilistic Property Language (PPL) in [4]. In this paper we present tools supporting PPL.

Existing work related to simulation based analysis of timing behaviour is given in [5,6]. Analytical methods for probabilistic analysis of timing behaviour is given in [7]. However, none of them fulfils our requirements completely, i.e. a rich and probabilistic modelling language and analyses scalable to large and complex systems.

Our contribution in this paper is a set of tools developed to support the ART Framework, how the tools can be used for impact analysis and regression analysis, and how ART-ML models can be validated. We also present how a company can benefit from using the ART Framework and our experiences from introducing parts of the framework at ABB Robotics in Sweden, one of the world's largest producers of industrial robots and robot control systems.

The paper is organized as follows: In Section 2 the design rationales behind the framework is discussed and in Section 3, we give a brief presentation of the ART Framework, including the modelling language ART-ML and the property specification language PPL. Further, in Section 4, we discuss the important topic of validating models. Section 5 describe the tools we have developed to support this framework. Section 6, present some of the the experiences we got from using the ART Framework in an industrial setting. Finally, the paper is concluded in Section 7.

2 Design Rationales

There exist many analytical methods for modelling and analysis of a real-time system's temporal behaviour [8,9]. However, the analytical models and analyses found in conventional scheduling theories are often too simple and therefore a real system cannot always be modelled and analyzed using such methods. The models used in those methods are not expressive enough in order to capture the behaviour of large and complex systems. There is no possibility of specifying dependencies between tasks and the models only allow worst-case execution times to be specified. Moreover, the analyses only cover deadline properties, i.e. whether or not every deadline is violated. In many real systems the temporal requirements are not only expressed in terms of deadlines, for instance there can be requirements on message queues such as starvation properties. Such requirements can not easily be verified with the analytical methods.

On the other extreme we find model-checking methods with rich modelling languages such as timed automata [10,11]. Timed automata allow modelling of temporal behaviour as well as functional behaviour. By using synchronization channels we can model dependencies between tasks in the system. However, model-checking does not scale properly to larger systems due to the state-space explosion which makes such an approach useless in a realistic setting.

Simulation is better from that point of view. Using simulation, rich modelling languages can be used to construct very realistic models, using e.g. realistic distributions of execution times. A disadvantage of the simulation approach is that we can not be confident in finding the worst possible temporal behaviour through simulation, since the state-space is only partially explored. Therefore, designers of hard real-time systems and safety-critical systems should not rely on simulation for analysis of critical properties if safe analysis methods are possible, such as model checking or scheduability analysis. However, when analysing a complex, existing system, that has not been designed for analysability, simulation is often the only alternative.

The design rationale behind the modelling language ART-ML is to provide a rich, probabilistic modelling language suitable for simulation. It should be possible to describe the behaviour of tasks, how the tasks synchronize and communicate and describe explicit execution times using probability distributions.

We could have used an existing notation, for instance timed automata, and extended it with execution time distributions and probabilities. However, we wanted a modelling language similar to the implementation, in order to facilitate the understanding of the model. We believe that software developers that are not used to formal methods are more likely to use this kind of models.

When designing the ART Framework we basically had one major trade-off to consider: being able to predict something at the cost of precision. We have chosen to use simulations as a tool for analysis. Even though a simulation might miss some situations, it may still point out potential problems and thus guiding the developers making the right decisions, while analytical methods are often not even possible to use in practice.

3 The ART Framework

The general idea in the ART Framework is the use of a model for analyzing the impact on timing and utilization of logical resources caused by maintenance operations, e.g. changing an existing feature or adding a new feature. It is also possible to use the tools in this framework to analyse measurements of the system directly, without using a model.

The core of the ART Framework is a general process (Figure 1) describing how model is constructed and used. Since the process is general it to be instantiated using any appropriate modelling and analysis method. The process is intended to be integrated in the life-cycle process at software development organizations.

We will start with a brief overview of the process and describe its steps in more details later on in this paper. The process consists of five steps:

1. Construct (or update) a structural model of the system, based on system documentation and the source code.
2. Populate the structural model with data measured on a running system. This data is typically probabilities of different behaviours and execution times.
3. Validate the constructed model by comparing predictions made using the model with observations of the real system. If the model does not capture the system's behaviour sufficiently, the first two steps are repeated in order to construct a better model and the new model is validated. This process should be repeated until a valid model is achieved. Model validation is discussed in Section 4.
4. Use the model for prototyping a change to the system, for instance if a new feature is to be added, the model is used for prototyping the change.
5. Analyse the updated model in order to identify any negative effects on the overall system behaviour, such as deadline misses or starvation on critical message queues.

If the impact of the change on the system is unacceptable, the change should be re-designed. On the other hand, if the results from the analysis are satisfactory the change can be implemented in the real system. The final step when changing

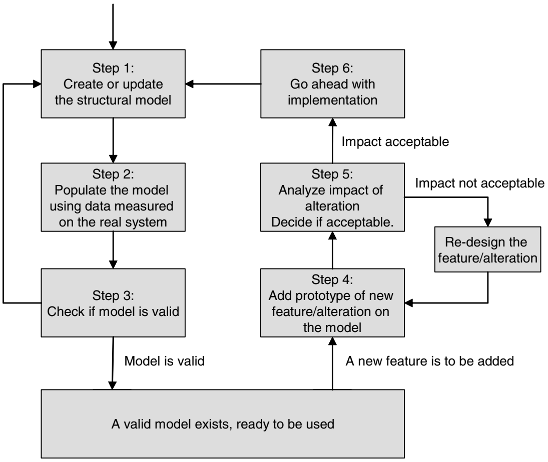

Step 1:
Create or update the structural model
Step 2:
Populate the model using data measured on the real system
Step 3:
Check if model is valid
Model is valid
A valid model exists, ready to be used
A new feature is to be added
Step 4:
Add prototype of new feature/alteration on the model
Step 5:
Analyze impact of alteration
Decide if acceptable.
Impact not acceptable
Re-design the feature/alteration
Impact acceptable
Step 6:
Go ahead with implementation

Fig. 1. The process of introducing and using a model for impact analysis

the system is to update the model so that it reflects the implementation in the real system by profiling the system in order to update the estimated execution times, steps 1-3 in Figure 1.

3.1 Constructing the Model

To construct a structural model of a system is to document the architecture and behaviour of the system in a notation suitable for analysis, at an appropriate level of abstraction. The resulting model describes what tasks there are, their attributes such as scheduling priority and their behaviour with respect to timing and interaction with other tasks using e.g. message queues and semaphores. To construct this model requires not only studies of the system documentation and code, but also to involve system experts throughout the complete process. This is important in order to select what parts of the system to focus the modelling effort on, since it is likely that some parts of the system are more critical than others and thus more interesting to model. Other parts of the system can be described in a less detailed manner.

Iterative reviewing of the model is necessary in order to avoid misunderstandings, due to e.g. different backgrounds and views of the system. If the system is large, this step can be tedious, several man months is realistic if the model engineer is unfamiliar to the system, according to our experiences. An experienced system architect can probably construct this structural model faster, but since such experts often are very busy, it is likely that the model developer is a less experienced engineer. In order to simplify the construction of the model, reverse-engineering tools such as Rigi [12,13] or Understand for C++ [14] can be used. These tools parse the code and visualize the relations between classes and files.

In step 2 of the process described in Figure 1, measurements are made in order to populate the model with execution times distributions and probabilities. The runtime behaviour of the system is recorded with respect to task timing, i.e. when tasks start and finish, how the tasks pre-empt each other, their execution times and the usage of logical resources such as the number of messages in message queues. This requires the introduction of software probes.

The output from the profiling is a stream of time-stamped events, where each event represents the execution of a probe. Typically, the execution of a probe corresponds to a task-switch or an operation on a logical resource (e.g. message queue).

Since the type of systems we consider are quite large and are changed often, we assume that the probes can remain in the system. This way we avoid the probe effect, since the probes becomes a part of the system, and also facilitates future measurements. The added probes will have a small effect on the system performance, less than one per cent, as the amount of probing necessary for our purposes is quite reasonable. The operating system used, VxWorks, allows user code to executed on every context switch, so by adding a single probe it is possible to record all context-switches and also the resulting scheduling status of the tasks. To record IPC communication requires four probes per task of intererst. This way we can record what message codes that are sent, i.e. what commands, but not the arguments. Recording the usage of a logical resource such as a message queue requires two probes per logical resource of interest. To register operations on semaphores could be done by adding three probes in the OS isolation layer.

Before the model is used for makeing predictions, it should be validated, i.e. compared with the real system in order make sure that the model describes the system behavior correctlty and in an apporpriate level of abstraction. In Section 4 we present a method for this.

3.2 The Modelling Language ART-ML

The ART-ML language describes a system a set of tasks and mechanisms for synchronization, where each task has an attributes part and a behaviour part.

The behavioural part of a task is described in an imperative language, C extended with ART-ML primitives. Both the task's temporal behaviour and, to some extent, its functional behaviour is modelled. The modeling primitives available are scheduling priority and how the task is activated, one-shot, periodically or sporadically. Moreover, ART-ML provides primitives and routines for sending messages to and receiving messages from message queues, as well as semaphore operations. The execution time for a block of code is modelled with a special statement that consumes time, `execute`. The execute-statement can also be used for modelling sections of code from the real system by their execution time only. Execution time is specified as a discrete probabilistic distribution in ART-ML.

An example of a task in ART-ML is:

```
TASK SENSOR
   TASK_TYPE: PERIODIC
   PERIOD: 2000
   PRIORITY: 1
BEHAVIOR:
   execute ((30, 200), (30, 250), (40, 300));
   sendMessage(CTRLDATAQ, MSG_A, NO_WAIT);
   chance(20){
      execute ((60, 200), (40, 230));
      sendMessage(CTRLCMDQ, MSG_B, FOREVER);
   }
END
```

The chance statement in ART-ML provides probabilistic selections, i.e. a non-deterministic selections based on probability. Chance express the probability of a particular selection. A chance statement can be used for mimicking behaviours observed in measurements of the real system, where the exact cause is not included in the model.

A message queue is a FIFO buffer storing messages. The message queue declaration contains the name and the size of the message queue. A message is sent to a message queue using the sendMessage and a message is read from a message queue using the recvMessage.

An ART-ML semaphore provides a mutual exclusion mechanism between tasks and conforms to the concept of the classic binary semaphores as proposed by Djikstra. A semaphore is locked using the sem_wait and released using sem_post routine.

3.3 Analysis of System Properties Using PPL

The analysis method decides what properties that can be analyzed and also affects the confidence assessment of the result. The main focus of the ART Framework is to support analysis of probabilistic system properties related to timing and usage of logical resources. For this purpose we have developed a property language called Probabilistic Property Language (PPL). This language was first proposed in [4]. We have recently implemented a tool for evaluating PPL queries, described in Section 5.3.

The analyses of system properties in our implementation of the ART Framework are based on trace of the dynamic behaviour, either from a real system implementation or from a simulation based on a ART-ML model. Based on the recording we evaluate the system properties specified in PPL.

System properties related to deadlines is a requirement on response times, either related to a particular task or features involving several tasks, i.e. *end-to-end response time*. A deadline property can be formulated as hard, or as a soft deadline. An example of a formulation of a soft deadline is that at least 90 % of the response times of the instances of a task are less than a specified deadline. In PPL, this property is formulated as follows:

```
P(TaskX.response < 1200) >= 0.9
```

Another property of interest could be the separation in time between instances of a task. The following PPL query checks if two consecutive instances of a task can be separated in time with less than 1000 time units.

```
P(TaskX(i+1).start - TaskX(i).end >= 1000) = 1
```

A PPL query may contain an unbounded variable. If replacing a constant value with an unbounded variable, the PPL tool calculates for what values the query result is true. The following PPL query finds the tightest deadline D that TaskX meets with a probability of at least 0.9.

```
P(TaskX.response < D) = 0.9
```

Resource usage properties are those addressing limited logical resources of a system such as fixed size message buffers and dynamic memory allocation. When analyzing such properties, the typical concern is to avoid "running out" of the critical resource. An example is the invariant that a message queue is always non-empty. In PPL, this is formulated as follows:

```
P(*.probe21 > 0) = 1
```

In this query above, it is assumed that the number of messages in the critical message queue is monitored using probe 21. The asterisk specifies that the condition should hold at all times. If a task name is specified instead, it means that the condition should hold when instances of that task starts.

3.4 Uses of the ART Framework

The ART Framework was initially developed for impact analysis, to predict the impact on timing and resource usage caused by a change. However, we discovered another use of the ART Framework, regression analysis, to analyse the current implementation and compare with previous versions. In this section these uses are described.

Impact Analysis. If a model has been constructed as described in Section 3.1 it can be used for predicting the impact an maintenance operation will have on the runtime behaviour of the system. The change is prototyped in the model and simulations of the updated model are made, generating execution traces. These are analyzed (as described in Section 3.3) in order to evaluate important system properties. This analysis can, in an early phase, indicate if there are potential problems associated with the change that are related to timing and usage of logical resources.

If this is the case, the designers should change their design in order to consume fewer resources. Since the change is not implemented yet, this means in practice to impose a resource budget on the implementer, specifying for instance a maximum allowed execution time.

If the impact of the change is acceptable, and is implemented, the model should be updated in order to reflect the implementation. This corresponds to step one trough three in the process shown in Figure 1, i.e. updating the model structure, profiling and validation.

The main problem with impact analysis is how to validate models. This is however not a problem unique for our approach, any formal analysis based on a model has this problem, if the model has been re-engineered from an existing system. In this paper we have presented a method for model validation (see Section 4), and in future work we intend to investigate other complementary methods.

Regression Analysis. Another use of the framework is regression analysis, i.e. to compare properties of the current release of the system with respect to certain invariants and with previous versions of the system. This is very close to regression testing, but instead of testing the functional behaviour, timing and resource usage are analyzed. It is also possible to compare the analysis result with earlier versions of the system. In this way, it is possible to study how the evolution of the system has affected the properties of interest. It might be possible to identify trends in system properties that could cause problems in future releases. If a model has been developed, the impact analysis can be used in order to predict how an extrapolation of a trend will affect the system.

In order to use regression analysis in a development organization, there is an initial effort of specifying the properties of interest, formulate them as PPL queries, define comparison rules and instrument the system with the appropriate software probes. The setup of the system should be specified in a document, in order to allow measurements to be reproduced. It is possible that different properties require different system setups, in that case multiple measurements of the system is necessary.

After this initial work, performing a regression analysis is straightforward and can be performed as one of many test-cases by a system tester, without deeper system understanding or programming knowledge. Measurements are made according to the documents initially produced. This results in execution traces, which are analyzed and compared with earlier releases, using a highly automated tool. Based on the comparison rules, the tool decides if there are alarming differences and informs the user of the outcome. A tool supporting this is presented in Section 5.3.

We are working on this approach in tight cooperation with ABB Robotics where we intend to introduce regression analysis. We have already integrated our recording functionality in their robot control system, which allows them to use our tools. The overall reaction among key persons at the company has been very positive. Before the method can be fully utilized at ABB Robotics, the relevant properties and the system setups used for the measurements must be specified.

4 Validation of Models

Validating a model is basically the activity of comparing the predictions from the model with observations of the real system. However, a direct comparison is not feasible, since the model is a probabilistic abstraction of the system. Instead, we compare the model and the system based on a set of properties, *comparison properties*. The method presented in Section 3.3 is used in order to evaluate these comparison properties, with respect to both the predictions based on the model and measurements of the real runtime system. If the predicted values of the comparison properties match the observations from the real system, the model is *observable property equivalent* to the real system. A typical comparison property can be the average response time of a task. It is affected by many factors and characterizes the temporal behaviour of the system.

Selecting the correct comparison properties is important in order to get a valid comparison. Moreover, as many system properties as practically possible should be included in the set of comparison properties in order to get high confidence in the comparison. The selected system properties should not only be relevant, but also be of different types in order to compare a variety of aspects of a model. Other types of comparison properties could be related to e.g. the number of messages in message queues (min, max, average) or pattern in the task scheduling (inter-arrival times, precedence, pre-emption).

Even if the model gives accurate predictions, there is another issue to consider, the *model robustness*. If the model is not robust, the model might become invalid as the system evolves, even if the corresponding updates are made on the model. Typically, a too abstract model tends to be non-robust, since it might not model dependencies between tasks that allow the impact of a change to propagate. Hence, it may require adding more details to the model in order to keep it valid and consistent with the implementation. If a model is robust, it implies that the relevant behaviours and semantic relations are indeed captured by the model at an appropriate level of abstraction.

4.1 Model Robustness

The robustness of a model can be analyzed using a *sensitivity analysis*. The basic idea is to test different probable alterations and verify that they affect the behaviour predicted by the model in the same way as they affect the observed behaviour of the system. Performing a sensitivity analysis is typically done after major changes of the model, in the validation step of the process. The process of performing sensitivity analysis is depicted in Figure 2. First a set of change scenarios has to be elicitated. The change scenarios should be representative for the probable changes that the system may undergo. Typical examples of change scenarios are to change the execution times of a task, to introduce new types of messages on already existing communication channels or change the rate sending messages. The change scenario elicitation requires, just as developing scenarios for architectural analysis, experienced engineers that can perform educated guesses about relevant and probable changes.

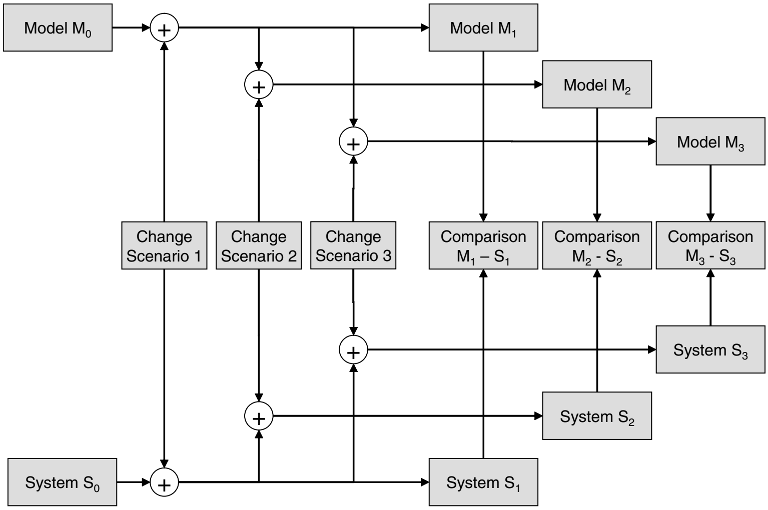


Fig. 2. The Sensitivity Analysis

The next step is to construct a set of system variants $S = (S_1, ..., S_n)$ and a set of corresponding models $M = (M_1, ..., M_n)$. The system variants in S are versions of the original system, S_0, where n different changes have been made corresponding to the n different change scenarios. The model variants in M are constructed in a similar way, by introducing the corresponding changes in the initial model M_0. Note that these changes only need to reflect the impact on the temporal behaviour and resource usage caused by the change scenarios, they do not have to be complete functional implementations. Each model variant is then compared with its corresponding system variant by investigating if they are equivalent as described in Section 3.3. If all variants are observable property equivalent, including the original model and system, we say that the model is robust.

5 Tools in the ART Framework

This section presents three tools within the ART Framework, supporting the process described in Section 3 (See [15] for a more detailed description of the tools).

- An ART-ML simulator, used to produce execution traces based on an ART-ML model.
- The Tracealyzer, a tool for visualizing the contents of execution traces and also allows PPL queries to be evaluated with respect to execution trace.
- The Property Evaluation Tool, PET, a tool for analysis and comparison of execution traces.

5.1 The ART-ML Simulator

The ART-ML Simulator has a graphical front-end with an integrated model editor, making it easy to use. This is not a simulator in the traditional sense, i.e. a general simulator application reading models as input. Instead, when the user clicks on the simulate-button, it translates the ART-ML model into ANSI C, compiles it using a standard C-compiler and links it with an ART-ML library. This results in an executable file, containing a synthesis of the ART-ML model. The synthesized model is executed for the specified duration, which produces an execution trace.

5.2 The Tracealyzer Tool

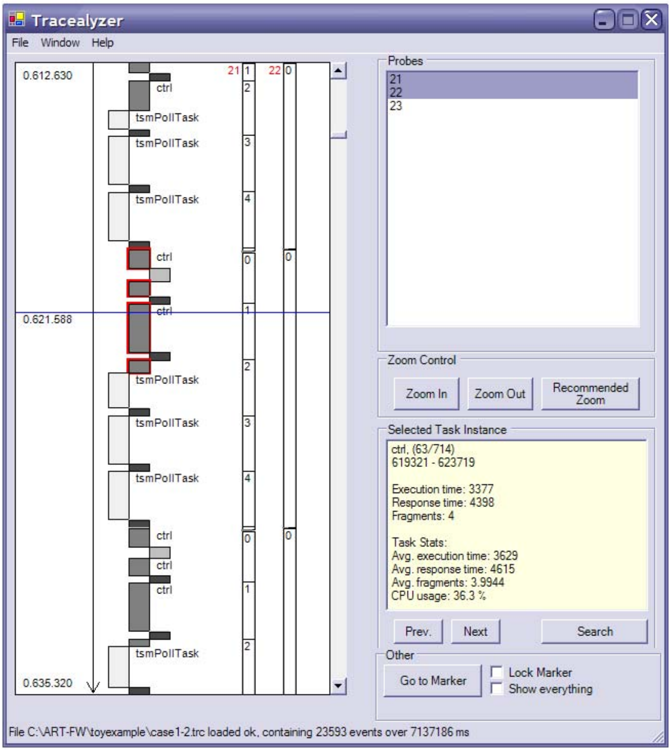
The Tracealyzer has two main features, visualization of an execution trace and a PPL terminal, a front-end for the PPL analysis tool. The execution trace is presented graphically. Tasks and generic probes are presented in parallel, allowing correlation between the task scheduling and the task behaviour. It is possible to navigate in the trace by using the mouse and also to zoom in and out and to search for task instances or probe observations with different characteristics. If the user clicks on a task instance, information about it is presented, such as the execution time and response time of the instance and the average execution and response times for the task. If more task statistics are desired, it is possible to generate a report, containing a lot of information about all tasks.

It is also possible to save a list of the task instances to a text file. This way, the data can be imported into e.g. Excel and visualized in other ways than the ones provided by the Tracealyzer. Apart from visualizing the data in an execution trace, the Tracealyzer also contains a PPL terminal. It is basically a front-end for the PPL analysis engine. The terminal contains two fields, one input where PPL queries can be typed and one output where the result is presented.

5.3 Property Evaluation Tool

The Property Evaluation Tool, PET, is a tool for analysing and comparing execution traces with respect to different system properties, formulated in PPL (described in Section 3.3). The operation of PET is rather simple. The user selects a file containing a predefined set of system properties, formulated as PPL queries. The file can also contain a comparison rule for each property, specifying what results that is acceptable and what is not. The user then only has to select the execution trace(s) to analyse.

If desired, two execution traces can be specified, but it is also possible to analyse a single trace. Specifying a second execution trace allows the tool to do an automatic comparison of the results using the comparison rules. When the user clicks on the analyse-button the properties are evaluated with respect to the trace(s) and the results are presented. If two traces are specified, the properties with comparison rules are compared automatically. Any properties where the rule has been broken are pointed out.



ctrl, (63/714)
619321 - 623719

Execution time: 3377
Response time: 4398
Fragments: 4

Task Stats:
Avg. execution time: 3629
Avg. response time: 4615
Avg. fragments: 3.9944
CPU usage: 36.3 %

File C:\ART-FW\toyexample\case1-2.trc loaded ok, containing 23593 events over 7137186 ms

Fig. 3. The Tracealyzer Tool

The application has three uses: impact analysis, regression analysis and model validation. In the impact analysis case, execution traces from simulation of two models are compared. One of the models is considered valid and used as reference. The other model contains a prototype of a new feature or other changes. By comparing these traces, the impact of the new feature can be analyzed.

When used for model validation, a trace from simulation is compared with a trace measured from the real system. This way, it is possible to gain confidence in the model validity.

In the regression analysis case, no data from simulation of models are used. Instead, execution traces measured from different versions of the system are analyzed and compared, in order to identify trends and alarming differences, which might be a result of undesired behavior in the system.

6 Benefits for Industry

If impact analysis can be performed when designing a new feature or other vast changes in the system, bad design decisions can be avoided. The designer of the feature can try alternative designs on a model and predict their impact on the system. This is likely to decrease maintenance costs since problems with timing and resource usage can be identified before implementation. Consequently, the time for identifying errors related to timing in late testing phases is reduced which decreases the cost for maintenance. This also leads to better system reliability.

Regression analysis and trend identification can point out undesired behaviour in the system that reflects in the system properties of interest, for instance execution times. It can also be used for performance analysis, by pointing out bottlenecks in the system. Information about trends in system properties can be used to plan ahead for hardware upgrades in the product which also is an important maintenance activity. If a trend in a property has been identified, impact analysis can be used to predict how the system will behave if a trend continues, for instance if a certain execution time keeps increasing as the system evolves.

The graphical visualization of execution traces, provided by the Tracealyzer tool is, according to our experiences, an effective way of increasing the understandability of the system. When the tool was introduced to developers and architects at ABB Robotics, showing them execution traces from the latest version of their system, we got immediate reactions on details and suspicious behaviours in the execution trace. We provided them with a new view of the run-time behaviour, increasing the understandability and facilitating debugging activities.

The results we have gotten so far from using the ART Framework at ABB Robotics indicates that maintenance costs can be reduced, as it enables impact analysis, regression analysis and significantly increases the system understandability. Even though the deployment of the framework is in an initial phase it has already pinpointed anomalies in the timing behaviour that were not previously known. Based on discussions with system architects, we believe that by deploying regression analysis we can reduce maintenance costs at ABB Robotics. As mentioned, we are working on introducing regression analysis in the company and later, when this has been used for a while, we plan to investigate the actual impact on maintenance costs.

We believe that introducing impact analysis could further reduce maintenance costs, as it helps system designers taking the right design decisions. Further research is however necessary in order to validate this approach.

7 Conclusions

In this paper we have briefly presented the ART Framework; the general ideas, the languages ART-ML and PPL. We have presented the three tools developed for this framework and an approach for validating ART-ML models. We have presented how the framework can be used for impact analysis, regression analysis and how the industry can benefit from these uses of the ART Framework. We have also presented our experiences from deploying parts of the framework in a development organisation, which strengthen our hypothesis that maintenance costs can be cut by introducing the methods proposed in the ART Framework.

We believe that this approach is very useful for its purpose, analysis of properties related to timing and resource utilization, targeting complex real- time systems. However, there is work remaining before we can validate this approach.

One problem with the approach described in this paper is the error-prone work of constructing the model. Instead of manually constructing the whole structural model, tools could be developed that mechanically generate at least parts of it, based on either a static analysis of the code, dynamic analysis of the runtime behaviour or a hybrid approach. This is part of our future work.

Further we intend to perform two case studies on the two uses of the ART Framework. In the first case study, we plan to further investigate the benefits and problems associated with deployment of regression analysis. We also intend to do a continuation of the case study on impact analysis, presented in [2], using a more advanced model and the tools presented in this paper. Later on, when this framework has been in use for some time, we plan to investigate how the maintenace cost at ABB Robotics have changed, by analysing the company's fault report database.

References

1. Schutz, W.: On the testability of distributed real-time systems. In: Proceedings of the 10th Tenth Symposium on Reliable Distributed Systems, (IEEE) 52–61
2. Wall, A., Andersson, J., Neander, J., Norström, C., Lembke, M.: Introducing Temporal Analyzability Late in the Lifecycle of Complex Real-Time Systems. In: Proceedings International Conferance on Real-Time Computing Systems and Applications. (2003)
3. Wall, A.: Architectural Modeling and Analysis of Complex Real-Time Systems. PhD thesis, Mälardalen University (2003)
4. Wall, A., Andersson, J., Norström, C.: Probabilistic Simulation-based Analysis of Complex Real-Time Systems. In: Proceedings 6th IEEE International Symposium on Object-oriented Real-time distributed Computing. (2003)
5. Audsly, N.C., Burns, A., Richardson, M.F., Wellings, A.J.: Stress: A simulator for hard real-time systems. Software-Practice and Experience **24** (1994) 543–564
6. Storch, M., Liu, J.S.: DRTSS: a simulation framework for complex real-time systems. In: Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium (RTAS '96), Dept. of Comput. Sci., Illinois Univ., Urbana, IL, USA (1996)
7. Manolache, S., Eles, P., Peng, Z.: Memory and Time-efficient Schedulability Analysis of Task Sets with Stochastic Execution Time. In: Proceedings of the 13nd Euromicro Conference on Real-Time Systems, Department of Computer and Information Science, Linköping University, Sweden (2001)
8. Audsley, N.C., Burns, A., Davis, R.I., Tindell, K.W., , Wellings, A.J.: Fixed priority pre-emptive scheduling: An historical perspective. Real-Time Systems Journal **8** (1995) 173–198
9. Liu, C.L., Layland, J.W.: Scheduling Algorithms for Multiprogramming in hard-real-time environment. Journal of the Association for Computing Machinery **20** (1973) 46–61
10. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126** (1994)

11. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a Nutshell. Springer International Journal of Software Tools for Technology Transfer **1** (1997)
12. Muller, H., Klashinsky, K.: Rigi: a system for programming-in-the-large. In: Proceedings of the 10th International Conference on Software Engineering. (1988)
13. Rigi Group: (Rigi Group Home Page) http://www.rigi.csc.uvic.ca/index.html.
14. Toolworks, S.: (Scientific toolworks home page) http://www.scitools.com/.
15. Andersson, J.: Modeling the Temporal Behavior of Complex Embedded Systems - A Reverse Engineering Approach. ISBN 91-88834-71-9. Mälardalen University Press (2005)

Static Timing Analysis of Real-Time Operating System Code

Daniel Sandell, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper

Dept. of Computer Science and Electronics, Mälardalen University
Box 883, S-721 23 Västerås, Sweden

Abstract. Methods for Worst-Case Execution Time (WCET) analysis have been known for some time, and recently commercial tools have emerged. However, the technique has so far not been much used to analyse real production codes. Here, we present a case study where static WCET analysis was used to find upper time bounds for time-critical regions in a commercial real-time operating system. The purpose was not primarily to test the accuracy of the estimates, but rather to investigate the practical difficulties that arise when applying the current WCET analysis methods to this particular kind of code. In particular, we were interested in how labor-intense the analysis becomes, measured by the number of annotations to explicitly constrain the program flow which is necessary to perform the analysis. We also make some qualitative observations regarding what a WCET analysis method would need in order to perform a both convenient and tight analysis of typical operating systems code. In a second set of experiments, we analyzed some standard WCET benchmark codes compiled with different levels of optimization. The purpose of this study was to see how the different compiler optimizations affected the precision of the analysis, and again whether it affected the level of user intervention necessary to obtain an accurate WCET estimate.

1 Introduction

A *Worst-Case Execution Time* (WCET) analysis finds an upper bound to the worst possible execution time of a computer program. Reliable WCET estimates are a key component when designing and verifying real-time systems, especially when real-time systems are used to control safety-critical systems like vehicles, military equipment and industrial power plants. WCET estimates are needed in hard real-time systems development to perform scheduling and schedulability analysis, to determine whether performance goals are met for periodic tasks, and to check that interrupts have sufficiently short reaction times [1]. However, WCET analysis has a much broader application domain; in any product development where timeliness is important, WCET analysis is a natural tool to apply.

Any WCET analysis must deal with the fact that a computer program typically has no fixed execution time. *Variations* in the execution time occur due to the characteristics of the software, as well as of the computer upon which

T. Margaria and B. Steffen (Eds.): ISoLA 2004, LNCS 4313, pp. 146–160, 2006.

the program is run. Thus, both the properties of the software and the hardware must be considered in order to understand and predict the WCET of a program.

The traditional way to determine the timing of a program is by measurements, also known as *dynamic timing analysis*. A wide variety of measurement tools are employed in industry, including emulators, logic analyzers, oscilloscopes, and software profiling tools [2,3]. This is labor-intensive and error-prone work. Even worse, it cannot guarantee that the true WCET has been found, since in general it is not possible to perform exhaustive testing.

Static timing analyses estimate the WCET of a program without actually running it. The analyses avoid the need to run the program by simultaneously considering the effects of all possible inputs, including possible system states, together with the program's interaction with the hardware. The analyses rely on mathematical models of the software and hardware involved. Given that the models are accurate enough, the result is a *safe* timing estimate that is greater than or equal to the actual WCET.

Recently, commercial WCET tools such as aiT [4] and Bound-T [5], have reached the market. However, practical experience of WCET analysis in industry has so far been rather limited, see Section 2.

In this case study we report from experiences when using a static WCET analysis tool to analyze code from the Enea OSE Real-Time operating system [6]. This is a commercial operating system, used in applications such as mobile phones and aircrafts, and thus an example of real production code.

Real-time operating systems are important to analyze with respect to timing properties, since they often are used in time-critical applications. Tasks with hard real-time constraints may make operating system calls. If no good WCET bound for the called code is known, then it is not possible to find a good WCET bound for the calling task either. Furthermore, OS services such as task switching must have good WCET bounds in a real-time system, since they also affect the timing properties of the application code. Finally, operating systems contain *disable interrupt regions* (or DI regions for short), where the interrupts are turned off, e.g., to provide a critical section where some shared resource is protected. The WCETs of such regions need also to be bounded, since their execution can delay higher priority tasks.

In our study, we analyzed some selected system calls and DI regions. We were somewhat interested in the precision of the analysis, but more in issues like how difficult it is to analyze typical operating systems code. WCET analysis cannot be completely automatic, (or we would have solved the halting problem), and manual user directives are typically needed to provide information that the analysis is not able to derive automatically. These directives provide problems: they can be erroneous, in which case the analysis might give an underestimation of the WCET, and providing the proper directives may be laborious and require a deep understanding of the code. This means that WCET analysis methods must be tuned to handle certain important classes of code with a minimum of needed user intervention. Our hypothesis was that operating systems code provides a particularly challenging class in this regard, and is much different in character than, for instance, signal processing code.

The second part of the study concerns how compiler optimizations affect the manual labor needed to perform an accurate WCET analysis. Optimizations may create a more complex and unstructured program flow in the resulting code, which may make it harder to provide proper annotations for constraining the program flow. In addition, we studied how compiler optimizations for speed and size, respectively, affected the WCET itself. This is also interesting: for instance, it is not evident that an optimization for average speed will give a lower WCET. For this part of the study we were not able to use the OSE operating system code, the reason being that this code, due to its low-level nature, only compiles with a small set of compilers with certain combinations of flags set. Instead, we used a set of standard WCET benchmarks.

The rest of this paper is organized as follows. In Section 2, we give a brief introduction to WCET analysis and related work in the area. In Section 3 the aiT WCET tool is described. Section 4 gives a short description of the OSE operating system. Section 5 presents the target processor for the analysis, including the associated development environment. Section 6 describes the experimental setup and the obtained results. Finally, in Section 7, we draw some conclusions and give ideas for further research.

2 WCET Analysis Overview and Related Work

Static WCET analysis is usually divided into three phases: a fairly machine-independent *flow analysis* of the code, where information about the possible program execution paths is derived, a *low-level analysis* where the execution time for atomic parts of the code is decided from a performance model for the target architecture, and a final *calculation* phase where flow and timing information derived in the previous phases are combined to derive a WCET estimate.

The purpose of the flow analysis phase is to extract the dynamic behaviour of the program. This includes information on which functions get called, how many times loops iterate, if there are dependencies between `if`-statements, etc. Since the flow analysis does not know the execution path which corresponds to the longest execution time, the information must be a safe (over)approximation including *all* possible program executions. The information can be obtained by *manual annotations* (integrated in the programming language [7] or provided separately [8,9]), or by *automatic flow analysis* methods [10,11,12]. The flow analysis is usually called high-level analysis, since it is often done on the source code, but it can equally well be done on intermediate or machine code level.

The purpose of low-level analysis is to determine the timing behaviour of instructions given the architectural features of the target system. For modern processors it is especially important to study the effects of various performance enhancing features, like caches, branch predictors and pipelines [13,14,15,16].

The purpose of the calculation phase is to calculate the WCET estimate for a program, combining the flow and timing information derived in the previous phases. A calculation method frequently used is IPET (Implicit Path Enumeration Technique), using arithmetical constraints to model the program flow and low-level execution times [8,17,12]. IPET calculations normally rely on integer linear programming to solve the generated constraint system.

Studies of WCET analysis of industrial code are not common. There are some reports on application of commercial WCET tools to analyze code for space applications [12,18,19], and in aerospace industry [20,21]. The experiences from some recent case studies are compiled in [22]. One of these case studies concerns WCET analysis for LIN and CAN communication software in cars [23].

An investigation of industrial embedded code has been done by Engblom [24]. He collected statistics of the number of occurrences of certain code features that may be problematic for a WCET analysis, like recursion, unstructured flow graphs, function pointers and function pointer calls, data pointers, deeply nested loops, multiple loop exits, deeply nested decision nests, and non-terminating loops and functions. In a more recent study [25], industrial code is investigated with respect to how amenable it is to a syntactical flow analysis, a method which detects certain loop patterns where iteration count bounds can be immediately given.

Studies of how to perform WCET analysis on operating system kernels are even more rare. We have done an earlier case study of the OSE operating system [26], where a number of DI regions were identified and analyzed. The study presented here is a followup. The only other work we know in the area is by Colin and Puaut [27]. They analyse some operating system functions of RTEMS, a small, open-source real-time kernel.

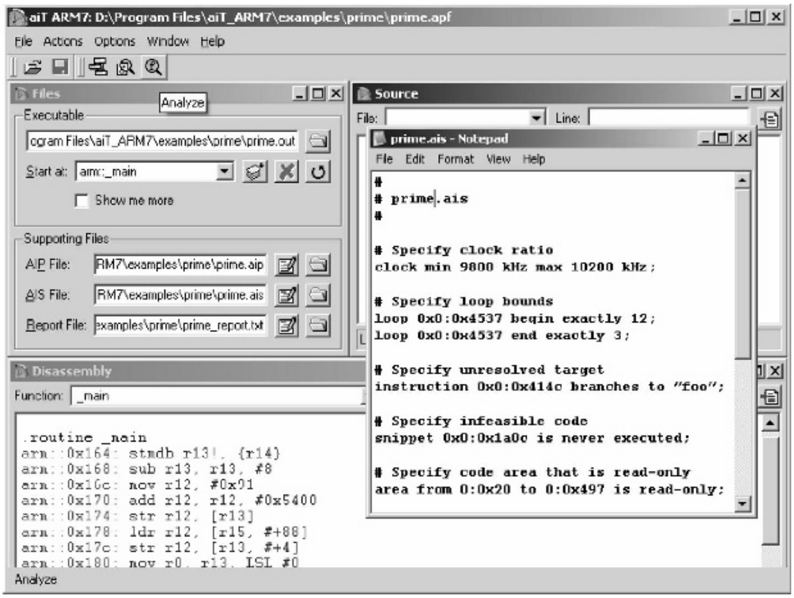
3 The aiT WCET Tool

The aiT tool is a commercial WCET analysis tool from AbsInt GmbH [4]. aiT analyses executable binaries, and it has support for a number of target architectures including the ARM7TDMI. aiT performs the following steps in its analysis:

- a *reconstruction of the control flow graph* from the executable code,
- an analysis to *bound loop iterations*, based on a combination of an interval-based abstract interpretation and pattern-matching tuned to the compiler that generated the analyzed code [28],
- a *value analysis* to determine the range of values in registers,
- a *cache analysis* that classifies accesses to main memory w.r.t. hits and misses, if the processor has a cache,
- a *pipeline analysis*, where a model of the pipeline behavior is used to determine the execution time of basic blocks, and finally
- a *path analysis* where an IPET calculation is made to determine the WCET.

In essence, the aiT WCET analysis conforms to the general scheme presented in Section 2. Several of the analyses in the chain are based on abstract interpretation [29], such as the value analysis and the cache analysis [17].

The aiT ARM7 tool analyses executables stored in .ELF format. This format contains information about the code, like symbol tables, which is used by aiT. This information is to some extent vendor-specific, meaning in reality that the


aiT ARM7: D:\Program Files\aiT_ARM7\examples\prime\prime.apf
File Actions Options Window Help
Files
Analyze
Source
Disassembly
prime.ais - Notepad
File Edit Format View Help
Analyze

Executable

ogram Files\aiT_ARM7\examples\prime\prime.out

Start at: arm::_main

□ Show me more

Supporting Files

AIP File: RM7\examples\prime\prime.aip

AIS File: RM7\examples\prime\prime.ais

Report File: examples\prime\prime_report.txt

File: Line:

Function: _main

```
.routine _main
arm::0x164: stmdb r13!, {r14}
arm::0x168: sub r13, r13, #8
arm::0x16c: mov r12, #0x91
arm::0x170: add r12, r12, #0x5400
arm::0x174: str r12, [r13]
arm::0x178: ldr r12, [r15, #+88]
arm::0x17c: str r12, [r13, #+4]
arm::0x180: mov r0, r13, LSL #0
```

```
#
# prime.ais
#

# Specify clock ratio
clock min 9800 kHz max 10200 kHz;

# Specify loop bounds
loop 0x0:0x4537 begin exactly 12;
loop 0x0:0x4537 end exactly 3;

# Specify unresolved target
instruction 0x0:0x414c branches to "foo";

# Specify infeasible code
snippet 0x0:0x1a0c is never executed;

# Specify code area that is read-only
area from 0:0x20 to 0:0x497 is read-only;
```

Fig. 1. aiT ARM7 WCET Tool Graphical Interface

aiT ARM7 tool can analyze executables from only a limited set of compilers. We used the ARM compiler, see Section 5, which belongs to this set.

The information present in the .ELF file and the executable itself is typically not sufficient to yield a good WCET bound for the analyzed code. In particular, information about program flow, such as bounds to loop iteration counts not caught by the loop bounds analysis, and knowledge of infeasible paths, has to be provided by the user. Therefore, aiT supports a set of *user annotations* to provide external information to the analysis [9]. Some of the more important annotations are: *loop bounds*, *maximal recursion depth*, *dead code*, and (static) *values of conditions*. The two latter can be used to exclude parts of the code, like error routines, which one may want to exclude from the WCET analysis even though their execution is feasible. In addition, there are a number of possible annotations to specify the control flow of subroutine calls, when needed, and to provide hardware-related information such as clock rate and address mapping to different kinds of memories.

Since aiT analyzes the binary code, the annotations have to be given on this level, which is cumbersome. For some compilers, aiT can use symbol tables to map annotations given for the source code to the binary level: however, the user must be aware that the control flow of the code might be different for the compiled code, which means that annotations valid for the source code might not be valid for the binary code. In our study, annotations were made on the binary level.

The larger lower window in Figure 1 illustrates the graphical interface for the aiT ARM7 WCET tool, including action keys for performing the WCET analysis and a subwindow with ARM7 assembler code extracted from the .ELF file. The front window gives some illustrative examples of possible annotations, including loop bound and dead code annotations.

4 The OSE Operating System

OSE is a real-time operating system, developed by Enea Embedded Technologies [6]. It is used in embedded system applications such as mobile phones and aircrafts. OSE is available for a number of target processors, mostly towards the high-end spectrum of embedded processors. The delta kernel of OSE, which has been used in this study, is available for ARM, StrongARM, PowerPC, Motorola 68k, and MIPS R3000. The OSE source code is written both in C and assembler.

The OSE kernels are process-based, fully preemptive, and provide priority-based process scheduling. Processes communicate through messages: messages are sometimes called *signals*, and are sent through buffers. Each buffer is identified by a signal number. Only a small number of system calls are needed to support most requests: for instance, basic interprocess communication can be handled by the two system calls `send` and `receive`.

In OSE, related processes can share the same memory pool. System calls like `alloc` and `free_buf` are used for memory handling by application processes.

Due to the communication and memory pool models, OSE must handle many shared memory structures. Since the process model is preemptive, the operating system code must contain quite a few critical sections to keep these structures consistent during updates. These sections are typically implemented by DI regions. Executing such a DI region can thus cause a temporary priority inversion, where a higher-priority process is delayed by a lower-priority process. It is therefore important to have small WCETs for the DI regions, in order to keep these delays down.

5 The ARM Board and Development Environment

We selected the ARM7TDMI processor as target architecture since it is widely used, since OSE is implemented for it, and since there is a version of aiT for it. This is a 32-bit RISC processor, with an uncached core and a 3-stage pipeline. Most of the instructions are executed in a single clock cycle.

Interrupts are enabled (EI) and disabled (DI) by setting some bits in a status register. This is done with a move-register-to-status-register instruction. Thus, an EI or DI will be executed depending on the contents in the source register.

5.1 ARM Development Tools

The ARM development toolkit[1] contains an ANSI C compiler, assembler, linker, ARMulator simulator, and an ARM development board. The C compiler produces ARM object format or assembly source output. It can be run with a variety of flags, including different levels of optimization for both space and execution time.

[1] We used ARM Developer Suite Version 1.2.

Table 1. Analyzed system calls

system call	description
`alloc`	Allocation of memory in a pool
`free_buf`	Free allocated memory
`receive`	Receive signal from another process
`send`	Send signal from process to another process

We used the ARMulator in our experiments. The ARMulator is a simulator, which makes it possible to evaluate the behaviour of a program for a certain ARM processor without using the actual hardware. The ARMulator model consists of four main components:

- The ARM processor core model that handles the communication with the debugger.
- The memory system. It is possible to modify the memory model, for instance w.r.t. different RAM types and access speeds.
- The coprocessor interface that supports custom coprocessor models.
- The operating system interface, which makes it possible to simulate an operating system.

It is possible to measure the number of clock cycles used by a program using the ARMulator. Bus and core related statistics can also be obtained from the debugger. There is no guarantee that the ARMulator timing model corresponds exactly with the actual hardware. However, since ARM7TDMI is an uncached and not very complex core, we expect the ARMulator to be rather cycle accurate.

6 Experimental Setup and Results

We made a series of experiments. First we analyzed a set of OSE system calls, and a number of DI regions in the OSE operating system, using the aiT tool. Then we investigated the influence of code optimization on WCET analysis. For this experiment, we compiled some standard benchmark programs with different levels of optimization, and performed WCET analyses on the resulting binaries, again using the aiT tool. For these binaries, we also tried to find the exact WCET by simulating the longest path with the ARMulator. This was done in order to estimate the accuracy and safety of the WCET estimates provided by the static analysis. In all experiments, we used the ARM C compiler, and we assumed a memory model with zero wait states for both the WCET analysis and the simulation (i.e., an instruction is executed in same number of clock cycles no matter where in the memory it is stored). The estimated WCET results and simulated execution times are given in number of clock cycles.

6.1 Analysis of System Calls

We analyzed the OSE system calls given in Table 1. These calls include error checks and use advanced memory protection. They are real-time classified system calls in OSE.

A problem that we soon discovered is that the execution time of these system calls depend on many parameters, such as the number of signal buffers, or

Table 2. Description of performed analyses and assumptions made

system call	restrictions of the analysis	assumptions
alloc(a)	Buffers of correct size exist	
alloc(b)	No buffers of correct size exist	No swap out handler is registered
free_buf	There are two pools in the system	
receive(a)	Receive all signals	The signal is first in the queue. No swap out handler is registered. A 20 bytes signal is copied. No redirection.
receive(b)	Receive a signal	The signal is at second place in the queue. Max 2 buffers before in the queue. No swap out handler is registered. A 20 bytes signal is copied. No redirection.
send(a)	Send a signal to a process with higher priority	The call to int mask handler is not analysed. No swap out handler is registered and the analysis stops before the interrupt process is called. No redirection.
send(b)	Send a signal to a process with lower priority	No redirection

Table 3. Result of system call analyses

system call	funcs	instr	blocks	loops	annot	WCET
alloc(a)	1	78	15	0	10	127
alloc(b)	9	390	54	0	18	433
free_buf	2	100	19	0	15	186
receive(a)	15	531	119	2	29	821
receive(b)	17	609	143	4	33	1469
send(a)	4	281	56	0	32	493
send(b)	5	288	62	0	33	417

maximal message sizes. Assuming a global worst-case scenario, where all these parameters assume their “worst” values, can give very poor WCET estimates for actual configurations, where these parameters typically are fixed, and assume much smaller values. Furthermore, one is often only interested in WCET estimates for the system executing in normal operation. This means that certain feasible paths, like error handling routines, may not be interesting to analyze.

We dealt with this problem in our experiments by assuming some “typical” scenarios for parameters affecting the WCET, like the number of buffers (after discussions with the OSE designers). Furthermore, we excluded typically uninteresting execution paths, like error handling, from the analysis by manual annotations, setting conditions to true or false or by explicitly excluding basic blocks. For `alloc`, `receive` and `send`, we assumed two different scenarios each. They are denoted by (a) and (b), respectively, in Table 2, which summarizes the conditions under which the analyses were made.

Table 3 gives the results of the analyses. For each analysed system call, **funcs** is the number of analysed routines in the call graph, **instr** is the total number of assembler code instructions, and **blocks** is the number of basic blocks. All these numbers are for the system calls with the error handling excluded. The estimated WCET's are given in column **WCET**.

The most interesting information in Table 3 is the number of annotations (**annot**) needed to perform each WCET analysis. As seen, quite a few annotations are required for each system call analysis. Another observation is that excluding the error handling yields significantly smaller code to analyze. For instance, `send` with full error check uses at least 39 routines. This indicates that

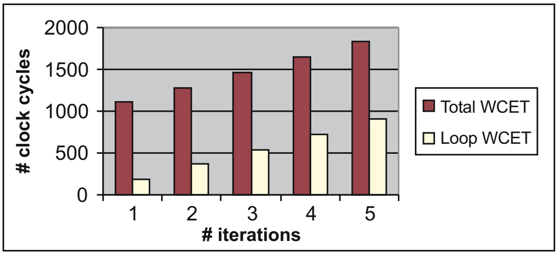


Fig. 2. `receive` WCET scaled with loop iterations

it really is important to identify and exclude execution scenarios that are not interesting to analyze, even if their execution is feasible.

Some of the analyzed system calls contained loops. Providing upper bounds for these loops posed a problem since almost all were dependent on dynamic data structures present in the system. As mentioned in Section 3, aiT performs a loop bound analysis. According to the aiT developers [28], their loop bound analysis typically bounds 70-95% of all loops automatically (ARM7 code compiled with the Texas Instrument ARM compiler). However, the loop bound analysis method for aiT cannot handle loops that depend on parameters not known at compile-time, making the recognition rate very low for the OSE system calls.

An illustrating example is a loop appearing in `receive`. The loop iterates through an array with signal numbers, searching for a specific signal buffer. Each iteration of the loop takes 13 clock cycles: thus, each iteration has a limited impact on the total WCET. However, there are more than 32000 possible signal numbers. In many system configurations, the actual number of signal numbers will be statically bound by a much smaller number, and the calculated WCET will be a huge overestimation if based on the maximal number of signal numbers allowed by OSE.

Another interesting loop is found in `receive`(b). The loop iterates through a queue of buffers, and the number of iterations is bounded by the number of buffers searched before finding the right one. Unfortunately, the number of buffers in the system may be hard to know statically since it depends on the current system state. The time for an iteration was 182 clock cycles and was a significant part of the total execution time of the system call. In Figure 2 we have tabulated the total WCET of `receive`(b) against the number of iterations performed in the loop. If the loop iterates more than five times, then its total contribution to the WCET will exceed the contribution of the rest of the executed code. The WCET for `receive`(b) in Table 3 was given under the assumption that at most two buffers were searched before the right buffer was found, which is correct in a scenario where the system only has two buffers.

The analysis of the system calls was done by the first author, who made the work as part of his M. Sc. thesis. Thus, he may be considered a typical engineer who has not yet acquired a lot of experience using WCET analysis tools, and who is not particularly knowledgeable about the code to be analysed. The analysis was quite labor-consuming, taking in total a few weeks to perform, even if the analyzed code in the end became quite small. The main reason for this was that

Table 4. Properties of some example DI regions

DI region	instr	blocks	loops	annot	WCET
DI92728-EI92752	6	2	0	1	12
DI74156-EI74216	16	4	0	2	29
DI82928-EI83088	28	9	1	6	331

he first tried to correct all the warnings that occurred in the analysis, e.g., set unresolved branches and loop bounds, before actually understanding what parts of the code that should be excluded from the analysis. Secondly, he had to rely on information from the OSE designers to give feasible loop bounds.

We conclude that it is possible to apply static WCET analysis to code with properties similar to the system calls in OSE Delta kernel. However, it is hard to fully automate the WCET analysis process on a 'one-click-analysis' basis. Instead, much manual intervention, and detailed knowledge of the analyzed code, is required to perform the analysis. Furthermore, if the obtained WCET values are to be useful, they must be calculated under the actual conditions for which the system is expected to run, with stronger bounds on system parameters and input arguments to system calls.

6.2 Analysis of Disable Interrupt Regions

In this experiment, we analysed 180 DI regions from the OSE operating system. This is a selection of the DI regions analysed in [26]. Most of the DI regions analysed were short and not so complex: 132 the regions contained five or less basic blocks, and only one of the selected regions contained a loop. Consequently, not so many annotations were needed and most DI regions needed only a few annotations: 119 of the 180 analyses needed two or less annotations. Figure 3 shows in detail how the number of annotations is distributed. In Table 4 the properties of three analysed DI regions are given together with their WCETs.

For this kind of code, the annotations were used mostly to restrict the WCET analysis to the actual program paths possible between the actual EI and DI operations. This is not always a trivial task, since DI regions may span function boundaries. Two different types of annotations were used for this. The condition annotation was used to follow the paths in the basic block graph, and the dead code annotation was used to make sure that the analysis would stop at the correct instruction. Only one loop, which is looking for any changes in a signal buffer, was found. It could not have its iteration count bounded automatically by aiT. Therefore, we manually set the loop bound to 10 to be able to extract a WCET estimate.

We conclude that DI regions are more suitable targets than the system calls for automatic WCET analysis. However, for some DI regions expert knowledge of the code is required to provide correct annotations, making it hard to make the analysis fully automatic.

6.3 WCET Analysis of Optimized Code

Compilers for embedded system can optimize for both speed and size. In many applications, such optimizations are important. Thus, it is interesting to study how these optimizations affect WCET analysis of the resulting code.

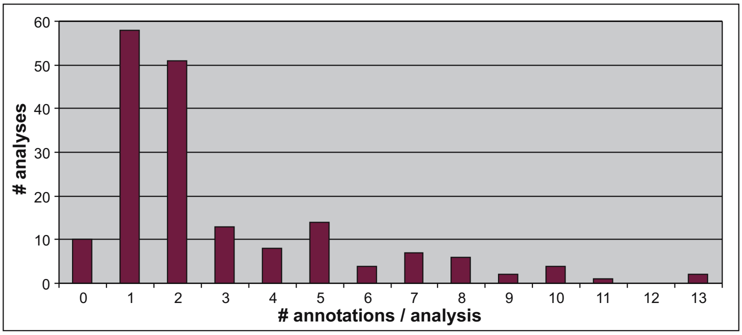


Fig. 3. Number of manual annotations per DI region

Table 5. Benchmarks for evaluating WCET and compiler optimizations

program	description	instr	blocks	loops
`bs`	Binary search for the array of 15 integer elements	28	10	1
`crc`	Cyclic redundancy check computation on 40 bytes of data	104	28	3
`expint`	Series expansion for computing an exponential integral function	50	18	2
`isort`	Insertion sort on a reversed array of size 10	42	7	2
`ns`	Search in a multi-demensional array	51	14	4
`select`	A function to select the nth largest number an array	91	34	4

The benchmarks used in this experiment contain conditional constructs. They are listed in Table 5 together with their size, numbers of blocks, and number of loops. The figures are for binary code compiled with the ARM C compiler, with medium optimisation for space.

Each benchmark was compiled with optimization for size and speed, respectively, and with medium and maximum optimisation levels for both. In Table 6, the results are given. For each optimization we give the WCET estimate produced by aiT (**aiT**), the simulated time obtained from the ARMulator (**armu**), the ratio in precent between these (**+%**), and the number of annotations (**ann**). We have also repeated the experiment with a ARM7 C compiler from IAR Systems with similar results, see [30] for details.

When the benchmarks were highly optimized, the structure of the programs changed a bit, but in most cases it was not so difficult to find the corresponding code and make the proper annotations. Changes that occurred were, for instance, that a function was moved inside the callers body, and the loop control could be changed to the end of the loop instead of the beginning. The most difficult changes to handle annotation-wise were when loop fission or loop fusion occurred.

Interestingly, the results indicate that it was not harder to perform an accurate WCET analysis for highly optimized code. Although the WCET estimate dropped with up to 48% when optimizing for speed, the ratio between WCET estimate and simulated execution time stayed quite constant. It was somewhat harder to produce annotations, but not much harder. (The number of annotations even drop some with increasing level of optimization, but this is mainly an effect of the code size decreasing with increasing levels of optimization.)

Table 6. How compiler optimizations affect WCET

pro-	speed – medium				speed – high				size – medium				size – high			
gram	aiT	armu	+%	ann	aiT	armu	+%	ann	aiT	armu	+%	ann	aiT	armu	+%	ann
bs	100	93	7.5	4	65	60	8.3	2	107	100	7.0	5	65	60	8.3	2
crc	34852	34804	0.1	7	27499	27455	0.2	7	34869	34821	0.1	7	27517	27473	0.2	4
expint	2208	1997	10.6	5	1150	1145	0.4	2	2263	2052	10.3	5	2113	1891	11.7	5
isort	1230	1190	3.4	4	1213	1190	1.9	3	969	962	0.7	4	944	919	2.7	3
ns	8518	8497	0.2	2	7228	7208	0.3	0	8601	8516	1.0	2	8603	8517	1.0	0
select	1357	1349	0.6	16	1333	1306	2.1	13	1428	1401	1.9	17	1362	1295	5.2	12

6.4 Justifying Obtained WCET Estimates

When comparing simulated ARMulator times and calculated aiT WCET estimates it should be noted that both methods rely on software models of the hardware. Therefore it is hard to say that one timing estimate is more correct than the other. Engblom [16] identifies several error sources when constructing hardware timing model, including hardware bugs, manual writing errors, and simulator implementation errors. Furthermore, for competitive reasons processor manufactures often keep the internals of their processor cores secret.

To get some justification of the quality of calculated WCET estimates we compared timing estimates from aiT and the ARMulator for a number of benchmarks (not included here, see [30] for details). The benchmarks contained features like system calls, loops and branches, but had only one single execution path through the program. By keeping track of the number of times each basic block was taken during a simulator run, we were able, by annotations, to provide exact bounds on the executions of each basic block for the WCET calculation. Thereby, the resulting timing discrepancies were not due to incorrect flow information, but only to differences in the hardware timing models.

The experiments showed that the aiT WCET estimates were on average about 5% larger than the times obtained using the ARMulator. For none of the tested benchmarks aiT gave a WCET lower than the timing produced by the ARMulator. The aiT developers have seen overestimations up to 4% for their ARM7 timing model, compared to measurements with a logic analyzer on a TI TMS470 bond-out chip [28]. We therefore conclude that the timing model of the ARMulator seems to be reasonably exact, or possibly providing slight underestimations on average. Thus, the WCET overestimations obtained when comparing with the ARMulator times should be quite similar, or possibly on average slightly overestimating the WCET overestimations for the real ARM7.

7 Conclusions and Future Work

The results indicate that static WCET analysis is a feasible method for deriving WCET estimates for real-time operating system code. For all analyzed parts of the OSE operating system we were able to obtain WCET estimates, including both system calls and DI regions.

We note however that the static WCET analysis technique is not yet mature enough to fully automate the timing analysis process on a 'one-click-analysis' basis. Instead, detailed knowledge of the analyzed code is required and often manual annotations must be provided.

Clearly, the usefulness of WCET analysis would improve with a higher level of automation and support from the tool. Especially important should be to develop advanced flow analysis methods, in particular to find more loop bounds automatically. For most of the loops analyzed in OSE the loop iteration bounds analysis of aiT would not produce a bound. Expert knowledge was needed to do this by hand, and the work was time-consuming and error-prone. Similarly, better support for easy exclusion of error handling routines from the normal WCET analysis would be of great value.

Another important conclusion is that absolute WCET bounds are not always appropriate for real-time operating system code. The WCET often depends on dynamic system parameters, like the number of signal buffers, whose absolute upper bounds may be large but which may be much more strongly bounded in actual configurations or running modes. An absolute WCET bound, covering all possible situations, may then provide a large overapproximation for that configuration or running mode.

Therefore, one would like to express the WCET conditionally, given that the system runs in a certain mode. Modes, or sets of modes, can often be encoded as value-range constraints on program variables (settings of flags, bounds on number of processes, etc.). Program flow constraints can also be expressed as value-range constraints, but on execution count variables. Thus, it seems interesting to develop means to communicate such information to the analysis in order to constrain the possible program flows for the given mode.

A parametric WCET analysis [31] may also be useful, especially for handling code like system calls. This type of WCET analysis could express how the WCET for system calls depends on, e.g., the system state and the input arguments.

Another conclusion is that the constant time often assumed for, e.g., context switch, in real-time scheduling theory, will be a large overestimation in many cases. A conditional WCET, in terms of system state and input arguments, would lead to a much tighter value, and thus a better utilisation of the system.

We have done some additional case studies of WCET analysis for industrial codes since the case study reported here was performed. The experience from these case studies largely confirms the conclusions drawn here [22].

Acknowledgements

This work was performed within the ASTEC competence center[2], under support by VINNOVA[3]. We want to thank AbsInt GmbH [4] for giving us access to the aiT tool, as well as Enea Embedded Technologies [6] for giving us access to source code and binaries for the OSE operating system.

[2] www.astec.uu.se

[3] www.vinnova.se

References

1. Ganssle, J.: Really Real-Time Systems. In: Proc. Embedded Systems Conference San Fransisco 2001. (2001)
2. Ive, A.: Runtime Performance Evaluation of Embedded Software. Presented at the 8^{th} Nordic Workshop on Programming Enviroment Research (1998)
3. Stewart, D.B.: Measuring Execution Time and Real-Time Performance. In: Proc. of the Embedded Systems Conference (ESCSF'2002). (2002)
4. AbsInt: AbsInt company homepage (2005) `www.absint.com`.
5. : Bound-T tool homepage (2006) `www.tidorum.fi/bound-t/`.
6. Enea: Enea Embedded Technology homepage (2005) `www.enea.com`.
7. Kirner, R., Puschner, P.: Transformation of Path Information for WCET Analysis during Compilation. In: Proc. 13^{th} Euromicro Conference of Real-Time Systems, (ECRTS'01), IEEE Computer Society Press (2001)
8. Ermedahl, A.: A Modular Tool Architecture for Worst-Case Execution Time Analysis. PhD thesis, Uppsala University, Dept. of Information Technology, Box 325, Uppsala, Sweden (2003)
9. Ferdinand, C., Heckmann, R., Theiling, H.: Convenient User Annotations for a WCET Tool. In: Proc. 3^{rd} International Workshop on Worst-Case Execution Time Analysis, (WCET'2003). (2003)
10. Gustafsson, J.: Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation. PhD thesis, Dept. of Information Technology, Uppsala University (2000)
11. Healy, C., Sjödin, M., Rustagi, V., Whalley, D.: Bounding Loop Iterations for Timing Analysis. In: Proc. 4^{th} IEEE Real-Time Technology and Applications Symposium (RTAS'98). (1998)
12. Holsti, N., Långbacka, T., Saarinen, S.: Worst-Case Execution-Time Analysis for Digital Signal Processors. In: Proc. EUSIPCO 2000 Conference (X European Signal Processing Conference). (2000)
13. Heckmann, R., Langenbach, M., Thesing, S., Wilhelm, R.: The Influence of Processor Architecture on the Design and the Results of WCET Tools. IEEE Proceedings on Real-Time Systems (2003)
14. Engblom, J.: Analysis of the Execution Time Unpredictability caused by Dynamic Branch Prediction. In: Proc. 8^{th} IEEE Real-Time/Embedded Technology and Applications Symposium (RTAS'03). (2003)
15. Healy, C., Arnold, R., Müller, F., Whalley, D., Harmon, M.: Bounding Pipeline and Instruction Cache Performance. IEEE Transactions on Computers **48** (1999)
16. Engblom, J.: Processor Pipelines and Static Worst-Case Execution Time Analysis. PhD thesis, Uppsala University, Dept. of Information Technology, Box 337, Uppsala, Sweden (2002) ISBN 91-554-5228-0.
17. Ferdinand, C., Martin, F., Wilhelm, R.: Applying Compiler Techniques to Cache Behavior Prediction. In: Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97). (1997)
18. Holsti, N., Långbacka, T., Saarinen, S.: Using a worst-case execution-time tool for real-time verification of the DEBIE software. In: Proc. DASIA 2000 Conference (Data Systems in Aerospace 2000, ESA SP-457). (2000)
19. Rodriguez, M., Silva, N., Esteves, J., Henriques, L., Costa, D., Holsti, N., Hjortnaes, K.: Challenges in Calculating the WCET of a Complex On-board Satellite Application. In: Proc. 3^{rd} International Workshop on Worst-Case Execution Time Analysis, (WCET'2003). (2003)

20. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and Precise WCET Determination for a Real-Life Processor. In: Proc. 1st International Workshop on Embedded Systems, (EMSOFT2000), LNCS 2211. (2001)
21. Thesing, S., Souyris, J., Heckmann, R., Randimbivololona, F., Langenbach, M., Wilhelm, R., Ferdinand, C.: An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software. In: Proc. of the IEEE International Conference on Dependable Systems and Networks (DSN-2003). (2003)
22. Ermedahl, A., Gustafsson, J., Lisper, B.: Experiences from Industrial WCET Analysis Case Studies. In: Proc. 5th International Workshop on Worst-Case Execution Time Analysis, (WCET'2005). (2005) 19–22
23. Byhlin, S., Ermedahl, A., Gustafsson, J., Lisper, B.: Applying Static WCET Analysis to Automotive Communication Software. In: Proc. 17th Euromicro Conference of Real-Time Systems, (ECRTS'05). (2005) 249–258
24. Engblom, J.: Static Properties of Embedded Real-Time Programs, and Their Implications for Worst-Case Execution Time Analysis. In: Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99), IEEE Computer Society Press (1999)
25. Sandberg, C.: Inspection of Industrial Code for Syntactical Loop Analysis. In: Proc. 3rd International Workshop on Worst-Case Execution Time Analysis, (WCET'2003). (2003)
26. Carlsson, M., Engblom, J., Ermedahl, A., Lindblad, J., Lisper, B.: Worst-case execution time analysis of disable interrupt regions in a commercial real-time operating system. In: Proc. 2nd International Workshop on Real-Time Tools (RT-TOOLS'2002). (2002)
27. Colin, A., Puaut, I.: Worst-Case Execution Time Analysis for the RTEMS Real-Time Operating System. In: Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01). (2001)
28. Lisper, B.: Personal communication with C. Ferdinand at AbsInt (2004)
29. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proc. 4th ACM Symposium on Principles of Programming Languages, Los Angeles (1977) 238–252
30. Sandell, D.: Evaluating Static Worst Case Execution Time Analysis for a Commercial Real-Time Operating System. Master's thesis, Mlardalen University, Vsters, Sweden (2004)
31. Lisper, B.: Fully Automatic, Parametric Worst-Case Execution Time Analysis. In Gustafsson, J., ed.: Proc. 3rd International Workshop on Worst-Case Execution Time Analysis, (WCET'2003), Porto (2003) 77–80

A Case Study in Domain-Customized Model Checking for Real-Time Component Software

Matthew Hoosier[1], Matthew B. Dwyer[2], Robby[1], and John Hatcliff[1]

[1] Kansas State University
Manhattan, KS 66506, USA
{matt, robby, hatcliff}@cis.ksu.edu
[2] University of Nebraska
Lincoln, NE 68588, USA
dwyer@cse.unl.edu

Abstract. Despite a decade of intensive research on general techniques for reducing the complexity of model checking, scalability remains the chief obstacle to its widespread adoption. Past experience has shown that domain-specific information can often be leveraged to obtain state-space reductions that go beyond general purpose reductions by customizing existing model checker implementations or by building new model-checking engines dedicated to a particular domain. Unfortunately, these strategies limit the dissemination of model checking across a number of domains since it is often infeasible for domain experts to build their own dedicated model checkers or to modify existing model checking engines.

To enable researchers to more easily tailor a model checking engine to a particular software-related domain, we have constructed an extensible and highly explicit-state software model checking framework called Bogor. In this paper, we describe our experience in customizing Bogor to check design models of avionics systems built using real-time CORBA component-based middleware. This includes modeling the semantics of a real-time CORBA event channel as a Bogor abstract data type, implementing a customized distributed state-space exploration algorithm that leverages the quasi-cyclic nature of periodic real-time computation, and encapsulating the Bogor checking engine in a robust full-featured development environment called Cadena that we have built for designing, analyzing, synthesizing, and implementing systems using the CORBA Component Model.

1 Introduction

Although focused research efforts have produced numerous techniques to reduce the storage and time required for model checking (c.f. [5]), general model checkers still are hampered by scalability limits. Certain universally applicable techniques such as partial order reductions are capable of cutting state space exploration costs dramatically—often by multiple orders of magnitude. Our and other investigators' past experience has demonstrated, however, that specialized knowledge about the *system domain* can be leveraged to enable further significant space

T. Margaria and B. Steffen (Eds.): ISoLA 2004, LNCS 4313, pp. 161–180, 2006.

reductions. In some cases, a new model-checking framework was developed targeted to the semantics of a family of artifacts [3, 12], while in other cases it was necessary to study an existing model checking framework in detail in order to customize it [4, 6].

Unfortunately, this level of knowledge and effort currently prevents many *domain* experts who are not necessarily experts in model-checking from successfully applying model checking to software analysis. Even though experts in different areas of software engineering have significant domain knowledge about the semantic primitives and properties of families of artifacts that could be brought to bear to produce cost-effective semantic reasoning via model checking, these experts should not be required to build their own model-checker or to pour over the details of an existing model-checker implementation while carrying out substantial modifications.

To enable researchers to more easily tailor a model-checking engine to a particular software-related domain, we have constructed an extensible and highly modular explicit-state model checking framework called Bogor [19].[1]

For treating realistic designs and implementations in widely-used languages such as Java and C#, Bogor provides rich base modeling language including features that allow for dynamic creation of objects and threads, garbage collection, virtual method calls and exception handling. For these built-in features, Bogor employs state-of-the-art reduction techniques such as collapse compression [15], heap symmetry [16], thread symmetry [2], and partial-order reductions. For tailoring to specific domains, Bogor provides (a) mechanisms for extending Bogor's modeling language with new primitive types, commands, expressions associated with a particular application, and (b) a well-organized module facility for plugging customized domain-tailored components into the model-checking engine. Moreover, Bogor is designed to be easily encapsulated within larger domain-specific development and verification environments.

In this paper, we report on a case study in which we used the above facilities to customize Bogor for checking properties of avionics system designs. In this domain, real-time event-driven systems are built from components defined in the CORBA Component Model (CCM) and deployed on Boeing's Bold Stroke middleware infrastructure built from a variety of real-time oriented services and the ACE/TAO real-time CORBA code base. Bogor is used to check global temporal properties of transition systems models formed from component transition systems as well as dedicated abstract domain models that capture the behavior of real-time middleware and services. Specifically,

- *new Bogor types* are defined for components, component ports, and events used for inter-component communication,
- *new Bogor modeling language commands and expressions* are introduced for declaring and connecting components and component ports, publishing and subscribing to events, and making calls on component method interfaces,
- *new Bogor internal modules* that model the complex semantics of the multi-layered ACE/TAO real-time event channel in which a pool of real-time

[1] Bogor project website: `http://bogor.projects.cis.ksu.edu`

threads is used to dispatch events to components and drive the computation of the system,
- *new state vector representations* are created that avoid storing data that does not change during execution in this domain (e.g., connection information for components),
- the general-purpose non-deterministic model-checker scheduler is replaced by a *new domain-specific scheduler* that implements the particular real-time rate-monotonic scheduling policy of the ACE/TAO real-time event channel of the Bold Stroke environment,
- the standard depth-first search algorithm of the model checker is replaced by a *distributed quasi-cyclic search algorithm* that takes advantage of the periodic nature of systems in this domain to achieve dramatic reductions in space and time.

This customized model checking framework has been encapsulated in a larger tool called Cadena[2] that we have constructed for supporting model-driven development and analysis of systems built from both industry standard component models such as the CORBA Component Model (CCM) and Enterprise Java Beans (EJB) as well as proprietary models such as Boeing's Prism component architecture. This required building translators to/from Cadena's design notations to Bogor's transition system notations.

In earlier work, we have described the design rationale, features, and implementation of Bogor [19] and Cadena [13], different approaches for modeling the ACE/TAO real-time event channel in Bogor [7], and the foundations of a sequential version of the quasi-cyclic search algorithm [9]. We reported in [13] our experience verifying some temporal properties of the variety mentioned here earlier. For example, we successfully verified systems to be free from race conditions which would corrupt the integrity of data displayed on a user interface; this data must find its way through a pipeline of components before being presented. In this paper, we focus on our experience in carrying out the tasks involved in the overarching goal of customizing Bogor to check real-time designs as part of Cadena, and we highlight the incorporation of the *distributed version* of the quasi-cyclic search strategy which significantly increases the scalability of the approach.

Compared to previous work on model-checking publish-subscribe architectures and component-based systems, our approach breaks new ground by using Bogor's sophisticated support for OO language features to capture more directly the structure of real-world component and middleware systems, by tailoring a variety of aspects of model-checking algorithms to the relatively complex threading model of the ACE/TAO real-time event-channel, and by incorporating the verification framework into a robust development environment that is being used by researchers at several industrial sites including Boeing, Lockheed-Martin, and Rockwell-Collins to develop realistic systems.

The rest of this paper is organized as follows. Section 2 provides a brief overview of the characteristics of systems in the domain that we are considering,

[2] Cadena project website:`http://cadena.projects.cis.ksu.edu`

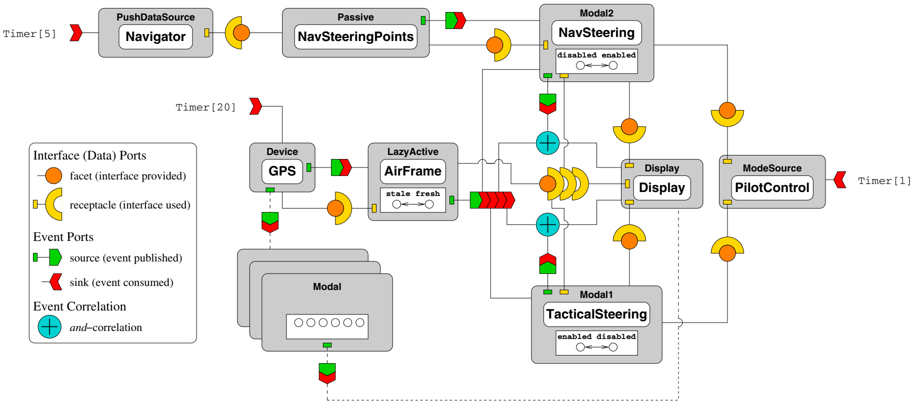


Fig. 1. Simple avionics system

Section 3 describes how Bogor's modeling language is customized to more directly support systems in this domain, and how Cadena design notations are translated into Bogor, Section 4 describes how modules of Bogor's built-in checking engine are replaced with customized modules that implement distributed quasi-cyclic search, Section 5 provides experimental results of applying the framework, Section 6 discusses related work, and Section 7 concludes.

2 Component-Based Avionics Systems in Cadena

We now give a brief overview of the structure of distributed real-time embedded systems that are designed using Cadena, and we explain how they give rise to quasi-cyclic state-spaces.

Figure 1 presents the CORBA component model (CCM) architecture for a very simple avionics system that shows steering cues on a pilot's navigational display. Although quite small, this system is indicative of the kind of system structure found in Boeing Bold Stroke designs. In the system, the pilot can choose between two different display modes: a *tactical* display mode displays steering cues related to a tactical (*ie*, mission) objective, while a *navigation* display mode displays cues related to a navigational objective. Cues for the navigation display are derived in part from navigation steering points data that can be entered by the navigator.

The system is realized as a collection of components coupled together via interface and event connections. Input position data is gathered periodically at a rate of 20 Hz in the GPS component and then passed to an intermediate AirFrame component, which in a more realistic system would take position data from a variety of other sensors. Both the NavSteering and TacticalSteering component produce cue data for Display based on air frame position data. The Navigator component polls for inputs from the plane's navigator at a rate of 5 Hz that are used to form NavSteeringPoints data. This data is then used to form navigational steering cues in NavSteering. PilotControl polls for a pilot steering mode at a rate of 1 Hz and enables or disables NavSteering and TacticalSteering accordingly. NavSteering and TacticalSteering are referred to as *modal components* since they each contain a *mode variable* (represented as a component attribute) whose value (enabled,disabled) determines the component's behavior. When a steering component is in enabled mode, it processes data from AirFrame and notifies the Display component that its data is ready to be retrieved over one of the modal component's interface connections. When a steering component is in disabled mode, all incoming events from AirFrame are ignored and the component takes no other action.

There are many interesting aspects to this system and its development in Cadena that we cannot explain here due to lack of space (see [7] for more details). We focus here on issues related to the quasi-cyclic structure of the state-spaces of these systems.

In Bold Stroke applications, even though at a conceptual level component event source ports are connected to event sink ports, in the implementation, event communication is factored through a real-time CORBA event channel. Use of such infrastructure is central to Bold Stroke computation because it provides not only a mechanism for communicating events, but also a pool-based threading model, time-triggered periodic events, and event correlation. In order to shield application components from the physical aspects of the system, for product-line flexibility, and for run-time efficiency, all components are *passive* (i.e., they do not contain threads) – instead, component methods are run by event-channel threads that dispatch events by calling the event handlers ("*push methods*" in CORBA terminology) associated with event sink ports. Thus, the event channel layer is the engine of the system in the sense that the threads from its pool drive all the computation of the system. The Event Channel also provides event correlation and event filtering mechanisms. In the example system of Figure 1, *and*-correlation is used, for instance, to combine event flows from NavSteering and AirFrame into Display. The semantics of *and*-correlation on two events e_1 and e_2 is that the event channel waits for an instance of both e_1 and e_2 to be published before creating a notification event that is dispatched to the consumer of the correlation.

Periodic processing in Bold Stroke applications is achieved by having a component such as GPS subscribe to a periodic time-out (e. g. `Timer[20]`) that is published by the real-time event-channel (the event-channel contains dedicated timer threads to publish such events). The time between two occurrences of a timeout of rate r is referred to as the *frame* of r (e.g., the length of the frame associated with the 5 Hz rate is 200 milliseconds).

In constructing transition system models of Bold Stroke applications, we take advantage of the fact that in rate-monotonic scheduling theory, which is used in Bold Stroke systems, the frame associated with a rate r can be evenly divided into some whole number of r'-frames for each rate r' that is higher than r. In the example system of Figure 1, the frame of the slowest rate (1 Hz) can be divided into five 5 Hz frames, and each 5 Hz frame can be divided into four 20 Hz frames. The longest frame/period (the frame associated with the lowest rate) is called the *hyper-period*.

We do not keep an explicit representation of clock ticks, but instead our model enforces the following constraints related to issuing of timeouts:

- a single timeout is issued for the slowest rate group in the hyper-period,
- timeouts for rate groups, r_i and r_j where $r_i > r_j$, are issued such that r_i/r_j timeouts of rate r_i are issued in a r_j frame.

These constraints determine the total number and relative ordering of instances of timeouts that may occur in the hyper-period. Combining these constraints with the scheduling strategy implemented in Bogor for Cadena models safely approximates all interleavings of timeouts and component actions (given the assumption that no frame overruns occur) [7].

Systems such as the one in Figure 1 are captured in Cadena using specifications phrased in three parts: (1) component behavioral descriptions describe the interface and transition semantics of component types such as the LazyActive component type in Figure 1 of which AirFrame is an instance, (2) a reusable model of a real-time CORBA event channel, and (3) system configuration information that describes the allocation of component instances and the port connections made between each of these instances as diagramed in Figure 1 (see [7] for a detailed explanation).

Note that in a real avionics system, there would be significant numeric computation to transform raw GPS data into a form that is useful for other components such as AirFrame. We do not represent this computation in our model for several significant reasons. First, in the actual systems supplied to us by Boeing, all such computation is stripped out for security reasons and to avoid dissemination of propriety information. Second, Boeing engineers are primarily concerned with reasoning about control properties associated with modes, and the data computations that are stripped out almost never influence the modal behavior of the system. In essence, Boeing engineers have by happenstance performed a manual abstraction of the system—an abstraction that produces a system that is very well-suited for model checking in that remaining mode data domains are finite and small.

3 From Cadena Scenarios to Bogor Transition Systems

3.1 Bogor Architecture for Customization

We have already asserted that Bogor is engineered in a manner intended to make introduction of new abstractions into the BIR language easy. Figure 2 presents a high-level view of its internal architecture, which separates loosely into three parts: (1) a front-end that parses and checks for well-formedness of a given model expressed in the BIR language, (2) interpretive components that implement the values and state transformations implied by BIR's semantics, and (3) model

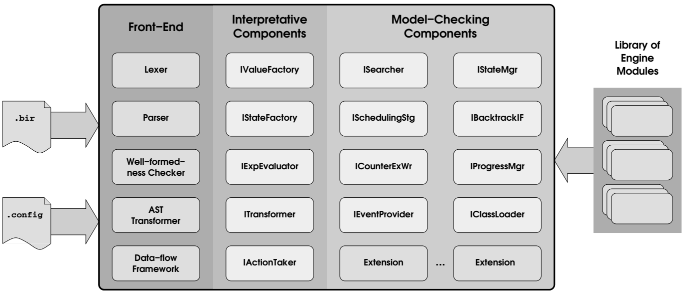


Fig. 2. Architecture and primary modules of Bogor

checking engine components that implement the core search algorithm and state storage strategies.

The separation of the various technical details required to implement any model checker (e.g., state-space search and state management) is achieved by presenting each component in an appropriate *design pattern* [10]. By employing these widely-used, well-documented programming idioms to hide irrelevant implementation details through encapsulation, dependency between components is reduced to only a minimal public API. An implementation of one component is, as a rule, not permitted to depend on any particular detail about another component's implementation.

The advantage of separating concerns into patterned modules is seen when replacing the default behavior of Bogor. The `ISearcher` state-space search module (a STRATEGY pattern), for instance, defines the general algorithm used to visit states. The default implementation uses a depth-first search where the left subtree of a state is fully explored before any node in a subtree to its right. A replacement of this entire algorithm with a breadth- first search, then, only required a different `ISearcher` which maintains an ordered queue of states to be explored.

One last aspect of Bogor's modular architecture has a tremendous payoff to the efforts we describe here: the bundles of Java code which implement the semantics of new abstract datatypes are contributed simply as new "extension" modules. Each contribution of a new datatype implements only the core API common to all Bogor modules, plus predictably named API methods corresponding to operators for the datatype. In the next section, we elaborate on this process with an example for the RT distributed middleware domain.

3.2 Re-usable Middleware for Event Propagation

Our domain customization of Bogor begins with the introduction of primitive types for components, ports, events in BIR (Bandera Intermediate Representation; the input language of Bogor). As seen in Fig. 3, we use the `typedef` keyword to inform Bogor that new, opaque datatypes are being added to the model. Then a rich set of operators for manipulating these types follow (we have pictured only

BIR specification

```
extension CAD for CADModule
{
   typedef Event;
   typedef Component;
   typedef Port;

   // constructor operator
   expdef CAD.Component
   createComponent(string name);

   // hook an object onto a component as
   // a named port
   actiondef
   registerPort(
      CAD.Component c, CAD.Port p,
      string name);

   // get all the subscribers to an event—
   // producing port
   expdef 'a[]
   getSubscribers<'a>(
      CAD.Component c,
      String eventPort);
}
```

Java implementation sketch

```
class EventValue implements
  INonPrimitiveExtValue { ... }

class ComponentValue implements
  INonPrimitiveExtValue { ... }

class PortValue implements
  INonPrimitiveExtValue { ... }
```

```
public class CADModule implements IModule
{
  ...

  public IValue createComponent(
    IExtArguments args)
  {
    ComponentValue result;
    ...
    return result;
  }

  public IBacktrackingInfo registerPort(
    IExtArguments args)
  {
    ComponentValue c = (ComponentValue)
      args.getArgument(0);
    PortValue p = (PortValue)
      args.getArgument(1);
    String name = ((IStringValue)
      args.getArgument(2))
          .getString();
    ...
  }

  public IValue getSubscribers(
    IExtArguments args)
  {
    ComponentValue c = (ComponentValue)
      args.getArgument(0);
    String port = ((IStringValue)
      args.getArgument(1))
          .getString();

    ...
    IArrayValue result = valueFactory
      .newArrayValue( ... );

    ...
    return result;
  }
}
```

Fig. 3. Declaration of middleware modeling abstractions (excerpts)

three here); as we have done, one usually provides an expdef constructor for each datatype and then a mixture of expdef side-effect-free expressions and actiondef state transformations on the datatypes.

Accompanying the BIR declaration of new modeling abstractions in Fig. 3 is a skeleton of the Java code which implements their semantics. The collection of operators provided by a namespace—corresponding to the BIR extension clause—is provided by a single Java class implementing Bogor's library type `IModule` (here, `CADModule`). To this, one typically contributes an additional Java class implementing the generic Bogor `INonPrimitiveExtValue` API for each new BIR type; in our case we add `ComponentValue`, `PortValue`, and `EventValue`. These classes supply methods for encoding the relevant state space values of their instances into the bit vector, traversing the heap, and other interpreter-related functions.

After the declaration and implementation of new datatypes is completed, we turn our attention to constructing models of component systems with them (see Fig. 4). First a `CAD.Component` BIR object is created for each real-world component. Above this, a `CAD.Port` instance is associated with a host component for each *provided* interface port in the component's static description. The extension also lifts event publication and subscribership to first-class citizenship by defining a set of operations (e.g., addSubscriberList() and addSubscriber()) to allow simple maintenance of the event propagation chain. By making interconnections among the `CAD.Port` instances (for facets and receptacles) and subscribership

```
CAD.Component EventChannel, GPS, AirFrame, NavDisplay, ... ;

EventChannel := CAD.createComponent();
GPS := CAD.createComponent("GPS");
AirFrame := CAD.createComponenet("AirFrame");
NavDisplay := CAD.createComponent("NavDisplay");

// create event-producing port
CAD.addSubscriberList(EventChannel, "timeOut20");
...
// connect EventChannel.timeOut20 to GPS.timeOut
{|tempComSub|} := new ComponentSubscriber;
{|tmepComSub|}.handlerFunction := EventHandlerType.{|common.BMDevice.timeOut<handler>()|};
{|tempComSub|}.portName := "timeOut";
{|tempComSub|}.component := GPS;
{|tempComSub|}.isSynchronous := false;
{|tempComSub|}.dispatchRate := 20;
CAD.addSubscriber<Subscriber>(EventChannel, "timeOut20", {|tempComSub|});
...
// create data-providing (interface) port
{|tempPort|} := CAD.createPort();
CAD.setPortMethodHandler<...>(
     {|tempPort|},
     "data<get>",
     {|common.ReadData.data<get>()|}.{|common.BMLazyActive.dataOut.data<get>()|});
CAD.registerPort(AirFrame, {|tempPort|}, "dataOut");

// connect consumer of data-providing port
CAD.connectPorts(Pair.create<...>(NavDisplay, "dataIn"), Pair.create<...>(AirFrame, "dataOut"));
```

Fig. 4. ModalSP system assembly using Bogor middleware primitives (excerpts)

records between event producers and sinks analogous to actual system deployment, we create an object-oriented BIR system which structurally mimics the real-world component software.

Figure 4 illustrates a series of calls to newly-defined middleware ADT operators used to assemble a fragment of a component model software system. As is the case for all Bogor extensions, each of these high-level API operations is implemented by a Java method on the extension's class definition. Each new abstract datatype introduced into BIR is implemented by a Java class. In order for Bogor to correctly distinguish among instances of an ADT, the developer must explicitly choose a bit pattern representative of the object's state, to encode into the model checker state vector. We leverage this by encoding only the sanitizied logical information about the component as described in the closing of Section 2: the values of any internal mode variables, external connection information, and subscriber list members. This allows an often dramatic reduction in the amount of storage required to maintain the seen-state set. By not encoding the minutia of a component ADT's implementation into the state vector, we also facilitate reductions by merging the state-vector representation of semantically identical but mechanically different data states.

The middleware datatype primitives for components, ports, and relatives are supplemented by BIR-language library routines (not shown) to simulate the action of an event channel in multiplexing event messages. During execution, a component *c*'s method publishes a message from a registered event source port by invoking the fireEventFromComponent() function, passing the name of the outbound event port as an argument. This library function performs the generic work of retrieving subscribership lists for the particular event source port on *c*, and then for each receiver of the event either directly invoking a handler function or queuing the message for later delivery if it crosses thread boundaries.

3.3 Component Behaviors as Transition Systems

The heart of Cadena component specifications are the intra-method Cadena Property Specification (CPS) *transition system specifications.* As discussed in Section 2, Cadena models focus on expressing mode-conditioned execution control, port method invocations, and storage/retrieval of data values rather than complicated numerical computations that are often present in avionics applications.

Client Port Method Invocations: The BIR language includes direct support for virtual functions, so achieving standard object-oriented method dispatch (e.g., when Bogor is used to model Java programs) is easy. Calling port methods on remote components requires an extra step of indirection beyond this; the receiving component and the interface method's virtual dispatch table key must be determined first. This is accomplished by introducing a wrapper function for calling each different interface virtual method. The calling component's method body invokes the wrapper function, passing as arguments the client component instance, client port name, and desired method name. The wrapper function then consults the port interconnection information registered (as seen in Figure 4) to retrieve the remote component instance and function reference. A virtual method call is made on the provider component.

Variable Uses: Simple private variables and mode variables may both be used in CPS expressions and as arguments to port method calls. These variables have persistent values and are allocated in a per-component-instance basis. The basic Bogor middleware extension module defines polymorphic getAttribute() and setAttribute() operations on the `CAD.Component` datatype. CPS variables are stored and retrieved directly using this API; l-value occurrences of a variable are converted to setAttribute updates, and r-value uses are replaced inline with getAttribute expressions.

Conditionals: CPS allows both standard if branches and case switches on mode variables. These are both translated directly as BIR guarded commands. One must take care to avoid inadvertently blocking a transition by failing to add a fall-through case for a missing else branch or default case; this is easily accomplished by adding an extra guarded transition at the furcation point whose enabling condition is the conjunction of every other branch's test condition negated.

3.4 Custom Scheduler for Reducing Interleavings

As explored in [7], the target environment for Cadena system designs—a real-time processor with tightly controlled scheduling policies—allows dramatic reductions by preventing the exploration of impossible thread interleavings. Cadena component code is executed by threads from a priority-based pool. Further, external verification techniques can decide whether a scenario configuration satisfies timing requirements (that is, whether frame overruns occur). Accordingly, all thread interleavings in which a lower-priority event dispatcher preempts a higher-priority thread are irrelevant. Similarly, any interleaving which corresponds to the system timer delivering a fresh batch of jobs before the current jobs are finished, is infeasible.

A custom Bogor scheduling module (replacing the default `ISchedulingStg` in Fig. 2) leverages these domain insights. Bogor's default scheduler module exhaustively traverses the set of enabled transitions given the current model state. Our distributed real-time scheduling module first retrieves the default scheduler's calculated set of enabled transitions, then deletes any transitions which correspond to priority inversion or frame overrun. This is done by using state introspection facilities in Bogor to perform a sophisticated analysis of priority information encoded in each thread's stack variables to make determinations about relative priorities of threads possible (without extensive static annotation of these threads).

3.5 Extending Bogor to Inform Domain-Specific Counterexamples

The modeling approach described here introduces several new abstract datatypes (e.g., the previously seen `CAD.Component`, `CAD.Event`, and `CAD.Port`, plus others for generic lists, queues, etc.). While it is possible to use custom types in Bogor without providing counterexample detail, the resulting error traces are devoid of all state information for the domain-specific modules. Presumably, this is undesirable since the ADT is central to the analysis being undertaken.

To facilitate the debugging and analysis of counterexample witnesses, the middleware ADT modules each implement a Java method which writes a series of schema-governed elements into the overall counterexample file (itself an XML document). This grammar used to encode a custom datatype instance's state information allows unrestricted nesting of heap and primitive types inside an extension type. This allows nested unfolding of any BIR objects "buried" inside a container ADT. Our hands-on experience shows this to be valuable when debugging the implementation of Bogor modules themselves.

3.6 Automatic Creation of Bogor Transition Systems

Cadena contains a model generator that automatically translates CPS transition system into BIR. Architecturally, the model generator is a large Visitor pattern implementation. By walking the structure of CPS syntax trees, the translator can produce appropriate BIR codes to simulate the dataflow behaviors (see Section 3.3) of each component method. These are packaged inside the BIR functions which make up the targets of virtual method calls. After creating the virtual method dispatch tables of Section 3.3, the generator then writes a system assembly phase (executed by the main thread before any others are forked) in which component instances and ports are allocated, inter-port connections are configured, and event dispatch queues are initialized as described in Section 3.2.

4 Distributed Quasi-Cyclic Search

The time and storage costs required to explore the state space of a BIR transition system representative of a Cadena model increase, at a high level, with the number of permutations among all components' mode variables. Nondeterminism in the CPS dataflow statements is the primary source of branching in the transition systems, since the domain-specific scheduling policy makes most context switches deterministic. We now present an overview of a hybrid search algorithm which leverages these observations to make the state space exploration partitionable and, therefore, distributable among many processors.

4.1 Quasi-Cyclic Search

A state transition system is defined to be an ordered quadruple $\langle S, s_0, E, \rightarrow \rangle$, where S is the set of all possible states, s_0 is the initial state, $E \subseteq S$ is the set of final states, and $\rightarrow \subseteq S \times S$ is the transition relation. $\rightarrow$ is neither required to be defined on all inputs nor is necessarily a function.

Each state in a transition system is defined to be one valuation of all the system's state variables V. Each state variable v is either an explicit variable (e.g., global data) or an implicit program counter variable.

A classical depth-first search (DFS) exploration of a transition system's *state space* (the set of all $s \in S$ reachable from the start state s_0 by a possibly empty sequence of applications of transitions in $\rightarrow$) accords no variables in a state's *vector* of values any special status. Indeed, the values of variables serve only to distinguish one state from another. By maintaining a global *seen state set*, the depth-first algorithm is able to detect cycles and prune the exploration of a state whose successors have already been visited.

The *quasi-cyclic search* (QCS) [9] is a strategy for coping with exploding memory requirements. It seeks to decompose the exploration of a transition system from one large global traversal into the traversal of many smaller, independent sub-state spaces. For example, event-driven systems feature an event-dispatching thread which blocks at a well-known control point waiting for work. At this point, a large chunk of its data structures (event queues, auxiliary lists) have predictable contents (likely: empty). Formally, we systematize this intuition by saying that a *transient* subset $T = \{t_1, t_2, \ldots, t_n\} \subseteq V$ of the system's state variables repeatedly takes on the distinguished values $\langle v_1, v_2, \ldots, v_n \rangle$.

In order to decompose an overall state space into independent regions, then, a QCS identifies those states for which T variables take on these distinguished values. To do this, a boundary-state predicate $p : S \rightarrow \mathbb{B}$ is defined so that

$$p(s) \iff (t_1 = v_1 \wedge t_2 = v_2 \wedge \ldots \wedge t_n = v_n) \quad .$$

```
Procedure QCS()
  $seen_g$ := $\emptyset$
  RegionDFS($s_0$)
  while ¬queueEmpty()
    $s_g$ := dequeue()
    $seen_g$ := $seen_g \cup \{s_g\}$
    $seen_t$ := $\{s\}$
    RegionDFS($s_g$)

Procedure RegionDFS($s$)
  $workSet$ := $enabled(s)$
  while $workSet \neq \emptyset$
    $\alpha \leftarrow workSet$.remove()
    $s' := \alpha(s)$
    if $p(s')$
      if $s' \notin seen_g \wedge \neg$inQueue($s'$)
        enqueue($s'$)
    else
      if $s' \notin seen_t$
        $seen_t$ := $seen_t \cup \{s'\}$
      pushStack($s'$)
      RegionDFS($s'$)
      popStack()
```

Fig. 5. Quasi-cyclic search algorithm

The state space exploration is accomplished by doing a hybrid DFS-BFS traversal. Beginning with the initial state s_0, a depth-first search proceeds as with classical state space exploration. The search is pruned whenever a state s satisfying $p(s)$ is reached. Rather than proceeding (past such an s) down such paths, each such s is recorded in a pending-work queue. Each state space tree beginning from a seed state and bounded by either terminal states, internal repeated states, or p-satisfying states is called a *region*. When each region's DFS terminates, a p-satisfying state s_b is retrieved from the pending-work queue and another DFS is initiated from s_b. This algorithm is given in Figure 5.

4.2 Identification of Transient and Long-Lived Variables

The size and placement of decomposed regions in a QCS state exploration depends entirely on the choice of p. The domain modeler must select conditions that accurately reflect a meta-"reset" condition. In a graphical user interface, a p which holds when there are no pending user input events and the main control loop is at its initial location is likely a good choice. For real-time systems with priority scheduling, the developer may select a p which is satisfied when all jobs are completed and the predictable delivery of work units is about to arrive (start-of-frame). Cadena systems are similar to both cases (both user input and frame boundaries are present). A quasi-cyclic search for Cadena uses a p which holds exactly when the following conditions occur:

- The system abstraction of time (which wraps after each hyperperiod) is at its initial value: 0;
- The thread designated to generate system timeouts is blocked (e.g., not currently creating a timeout); and
- All event queues are empty and each event-dispatching thread is blocked awaiting work.

```
PROCESS COORDINATOR ( )
   ⟨ $seen_g$ := $\emptyset$
     $tasks$ := 0
     enqueue($s_0$)
   ⟩
   ⟨await queueEmpty( ) $\wedge$ $tasks = 0$⟩

PROCEDURE GETSEED ( )
   while true
      ⟨await ¬queueEmpty() $\vee$ $tasks = 0 \rightarrow$
         if ¬queueEmpty()
            $s_{temp}$ := dequeue( )
            if $s_{temp} \notin seen_g$
               $seen_g$ := $seen_g \cup s_{temp}$
               $tasks$ := $tasks + 1$
               return $s_{temp}$
            else
               continue
         else
            return nil
      ⟩

PROCESS REGIONSEARCHER ( )
   while true
      $s$ := GETSEED ( )
      if $s$ is nil
         break
      $seen_t$ := $\{s\}$
      REGIONDFS($s$)
      ⟨$tasks$ := $tasks - 1$⟩

PROCEDURE REGIONDFS ( $s$ )
   $workSet$ := $enabled(s)$
   while $workSet \neq \emptyset$
      $\alpha \leftarrow workSet$.remove( )
      $s' := \alpha(s)$
      if $p(s')$
         <enqueue($s'$)>
      else
         if $s' \notin seen_t$
            $seen_t$ := $seen_t \cup \{s'\}$
            pushStack($s'$)
            REGIONDFS($s'$)
            popStack( )
```

Fig. 6. Distributed quasi-cyclic search algorithm

Intuitively, these conditions correspond to the beginning of a new hyperperiod with no frame overrun. The long-lived values such as mode variables and event correlation automata are not tested by p (and are thus in the *non-transient* set) because they influence the inter-frame behavior. We remark at this point that no choice of p can result in an incomplete or incorrect state space exploration; the selection of a transient variables set T only affects the performance of QCS.

4.3 Adapting QCS to Distributed State Space Exploration

Because the region searches conducted by the REGIONDFS procedure in Figure 5 do not rely on any external data (except the global seen-before boundary state set), the algorithm is amenable to parallelization. This is done, in broad terms, by making the old REGIONDFS into an active process. Figure 6 gives an adaptation of standard QCS to a distributed environment. We have used angle brackets ("⟨" and "⟩") to denote critical section code running in mutual exclusion, in the manner of SyncGen specifications [8]. A distinguished instance of the COORDINATOR process maintains the global seen-before boundary state set $seen_g$, the counter of active tasks (*tasks*), and pending-work queue. As many REGIONSEARCHER processes as desired are initiated to consume boundary states from $seen_g$ and explore the decomposed state space one region at a time.

4.4 Architecture

We have adapted Bogor to the distributed version of QCS by writing a new state graph traversal module and substituting it for the standard searcher in the REGIONSEARCHER client process. Each of the subordinate client processes requests a unit of work (by invoking the equivalent of what we have shown as GetSeed), which is a tuple $< b, c, s >$ where b is the BIR transition from which the region to be checked is excerpted, c is the Bogor system configuration (runtime options, etc.), and s is the seed state (the formal parameter s of the RegionDFS procedure). The subordinate process constructs a complete Bogor system as directed by c, uses the lexical frontend to load the BIR system b, and finally starts a model check from state s which is bounded by the predicate p (this is encoded inside c). As the model check proceeds, each p-satisfying state encountered is reported back to the Coordinator process.

4.5 Transporting State Information

Bogor is implemented in Java. Fortunately, the Remote Method Invocation (RMI) subsystem directly supported in Java provides an effective mechanism for copying state variable data to and from the Coordinator process. A deep clone of the Bogor object containing all state information is automatically serialized and reconstructed on the remote host during an RMI call. All that remains to acclimate the state object to its new Bogor host process. An enhancement to the main Bogor state API allows the state object to re-acquire references to any required runtime modules.

4.6 Facilitating Counterexamples

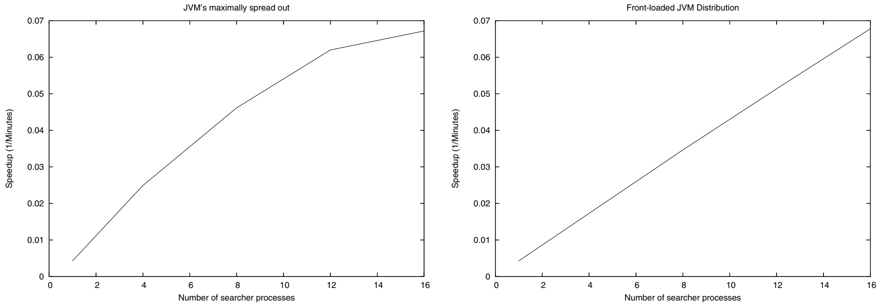
Constructing counterexamples from a standard DFS is simple: a simple examination of the backtracking information stack reveals the complete path to a violating state. In QCS, there is no “current” path from the root state to a property violation. The $seen_g$ set contains a series of p-states whose successors have been explored. Our implementation supplements this with two auxiliary lookup tables: (1) a map between each p-satisfying state and parent seed state, and (2) a map from each p-satisfying state and the series of backtracking steps which lead to the parent seed state. When a violating state is reached, all backtracking path fragments are concatenated to form a complete trace.

5 Results

To evaluate the scalability of our parallel quasi-cyclic search, we have chosen an updated version of a suitable system from the original experiments run on quasi-cyclic systems [9].

The specific model we chose to test is the result of automatically compiling the Cadena input artifacts for a ModalSP scenario *plus* two additional modal components, into representative BIR code. In the notation of the scenario configurations used in [9], this BIR system would be called a $\langle 2, 2, 2\rangle$ configuration.

The experimental platform consisted of 4-processor AMD Opteron 842 systems running at 1.6 GHz with 16 GB of memory each, running SuSE Linux 8.2 (beta release for AMD64). The nodes are connected with a 1000 Mbps copper



(a) Adding each new client to least-loaded machine

(b) Adding one fully-loaded client machine at a time

Fig. 7. Scalability graphs

ethernet switch. Bogor ran on top of the 1.5.0-beta-b32c native AMD64 Java Virtual Machine from Sun. Client processes were allocated 8 GB maximum heap size (half the total physical memory of its host).

Five total machines were used; in every instance one was reserved exclusively for the coordinating server; the remaining four were used to execute client processes.

Our initial series of trials was done by running $\frac{n}{4}$ out of n total client processes on each machine. Notice that the performance gains drop off steeply after 8 client processes are allocated in series (a) of Figure 7. This was something of a mystery to us, because the CPU and I/O load on the coordinator server never reached high levels. Why could our system only weakly support additional parallelism?

After careful examination, we noticed that the turning point in the performance curve on (a) corresponded to the point where the total size of all JVM heaps allocated on the client processes equal the amount of physical RAM available on the host. That is, the JVM's were consuming $\frac{8}{4} \times 8$ GB= 16 GB per host at this point. Increasing the total number of client processes past 8 implied double-booking the physical RAM on the hosts. Although the black-box nature of the Java Virtual Machine's memory manager prevents a detailed analysis, we surmise that memory contention prevented more than 2 client processes from fully utilizing the processor time on any one machine.

To test this hypothesis, we re-ran our test series. As before, additional client processes were added in groups of 4. This time, though, each new 4-processes client group was embodied as a fully loaded machine. This series, (b) in Figure 7, shows the performance gains of adding one *fully utilized* client machine at a time. The speedup curve is virtually linear.

Despite some problems getting full utilization of each client machine when very high RAM allocations are desired, our architecture for distributed quasi-cyclic searching appears to be intrinsically quite scalable. This should continue to be the case so long as (1) the cost of copying states is dominated by the time to search an average region and (2) the models themselves have sufficient nondeterminism to usually keep at least as many states in the unexplored-boundary-state set as there are client processors.

We have additionally run many smaller models (also generated by Cadena) through the distributed QCS system (BasicSP, standard ModalSP, MediumSP, etc.). While less systematic than the data presented here and less persuasive because their running times are much lower, these transition systems exhibited roughly the same performance improvement curves as additional client processes were brought online.

6 Related Work

Garavel et al. present a technique for distributing an explicit state space exploration across multiple network computers in [11]. By carefully choosing a hash function to use for partitioning states uniformly across network nodes, and by implementing non-blocking SEND and RECV operations between model-checking processes, Garavel et al. improved the scalability of a SCSI subsystem model to near-linear [11]. Our work on quasicyclic search does not use a static partitioning; rather a central pool of pending seed states is maintained and each node retrieves one as it becomes free. Additionally, we can afford to use synchronous analogues of the SEND and RECV operations because intra-region searches typically execute very long sequences of transitions before interrupting to perform network operations. We do not believe network messages impose a noticeable bottleneck on sytem performance. The Java PathFinder (JPF) model checker [18] uses a dynamic partitioning scheme in which the state-to-host hashing function changes when analyses indicate that the ration of network messages to transition execution could be improved by redefining the state partitioning scheme. Significant gains are made versus the baseline JPF static partitioning method. As mentioned in context of [11] above, the quasicyclic search instead reduces inter-node network traffic by decomposing a state space into many *independent* regions; the entirety of each is explored by one model checker process without network traffic interruptions.

Jones and Mercer improve on the static partitioning typically used in distributed model checking by *randomly* choosing the next state to expand from a set of priority-sorted frontier states [17]. In some cases where a Bayes heurestic search algorithm required many transitions to reach a first error state, introducing randomness into the search order significantly reduced the number of transitions executed before reaching a first error state. Our quasicyclic search in general seeks to rely on domain knowledge to improve parallel search performance (Jones and Mercer specifically attempt to develop model-independent methods), but in principle the approach of [17] could be used to inform intra-region searches.

Ben-David et al. in [1] and later Heyman et al. in [14] give algorithms for achieving load-balancing across the network nodes during model checking. The former achieved counterexample generation without requiring any one network node's process to contain a complete state set and significantly reduced overall memory requirements. The latter builds on this approach by using adaptive partitioning of the state space and compact Binary Decision Diagrams (BDD's) which can be transferred over a network, to further decrease memory requirements. Because these approaches are adaptations of the symbolic model checking algorithm, it is an open question whether they can be applied to our setting; the Bogor framework uses an explicit, domain-tailored state-space exploration algorithm.

7 Discussion

By leveraging various properties of the Bold Stroke avionics domain via customization of Bogor, we have been able to reduce the cost of model checking by multiple orders of magnitude compared to our earlier attempts [13] to model these systems in dSpin. That is, we manually introduced an appropriate abstraction of the underlying middleware specifically used in Bold Stroke. Ideally, we would like to have an automatic abstraction process, however, even with the current state-of-the-art abstraction techniques, it is still hard to extract an abstract model from CORBA middleware implementations (consisting of hundreds of thousands of lines of C++ code) that is coarse such that it is tractable, yet, precise enough to reason about the properties that we are interested in.

While the customization points of Bogor ease the incorporation of domain knowledge to reduce the cost of model checking, they have to be used with caution to ensure that the state-space exploration is still sound with respect to the properties being checked. For example, when incorporating a queue data structure in Bogor, one needs to ensure that its implementation (for example, its bit-vector encoding) is correct. This can be done by using unit testing on the linearization algorithm of the data structure. Moreover, its abstract operations such as enqueue and dequeue should satisfy the expected properties such as first-in first-out ordering. This can be checked by creating generic environments that exercise the abstract operations as BIR models, and use Bogor to check the properties, which can be expressed using automata. While these approaches do not prove the correctness of the customizations, they help ensure a certain level of confidence, and we gain more trust as the customizations are used frequently over time. In some sense, this is analogous to implementing a compiler optimization inside an optimizing compiler framework. That is, Bogor provides a framework for optimizing state-space exploration, however, each customization implemented in the framework has to be determined sound before being used.

As a complete beginner to Bogor, the first author required—all told—perhaps two weeks to overhaul and debug all of the approximately 15 new native BIR datatypes used in the Cadena models. The algorithmic extensions required to implement a quasi-cyclic search required approximately another week; this happened somewhat later when the author was better acquainted with Bogor. Still, even though we believe Bogor is much easier to customize than many existing model checkers, a fair amount of knowledge of the internals of a model checker is sometimes required for such efforts. The Bogor website provides a variety of educational, tutorial, and example materials to ease the burden of learning the infrastructure, and we continue to search for additional automated techniques to aid developers in building, testing, and debugging their own Bogor extensions. Bogor has been used to teach graduate level model checking courses at a number of institutions in North America (e.g., Kansas State University, University of Nebraska-Lincoln, Brigham Young University, Queen's University), and Europe (e.g., University of Göttingen), and multiple model checking researchers outside of our research group have effectively customized Bogor to particular domains.

We have been able to introduce model checking concepts to several different industrial research groups at Boeing, Lockheed-Martin, and Rockwell-Collins due to the incorporation of Bogor into Cadena. However, to date, engineers at these sites tend to make much more use of the design and structural analysis capabilities of Cadena rather than the model checking technology because they often do not want to put forth the effort required for writing transition models of components. To address this issue, we are exploring (a) techniques for inferring component transition models from run-time trace data and (b) opportunities for further leveraging component transition system models in other forms of analysis and code synthesis.

Acknowledgments

The authors would like to thank the members of the Santos group at Kansas State University for many useful discussions about issues related to this paper, especially Jesse Greenwald, Georg Jung, and Xianghua Deng. The research reported in this paper was supported in part by the U.S. Army Research Office (DAAD190110564), by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), by NSF (CCR-0306607) by Lockheed Martin, by Rockwell-Collins, and by an IBM Corporation Eclipse Award.

References

1. S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In *Proceedings of Formal Methods in Computer-Aided Design*, pages 390–404, 2000.
2. D. Bosnacki, D. Dams, and L. Holenderski. Symmetric SPIN. *International Journal on Software Tools for Technology Transfer*, 2002.
3. G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – a second generation of a Java model-checker. In *Proceedings of the Workshop on Advances in Verification*, July 2000.
4. W. Chan, R. J. Anderson, P. Beame, D. H. Jones, D. Notkin, and W. E. Warner. Optimizing symbolic model checking for statecharts. *IEEE Transactions on Software Engineering*, 27(2):170–190, February 2001.
5. E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.
6. C. Demartini, R. Iosif, and R. Sisto. dspin : A dynamic extension of SPIN. In *Proc. of the 6th International SPIN Workshop*, volume 1680, pages 261–276. Springer-Verlag, Sept. 1999.

7. W. Deng, M. B. Dwyer, J. Hatcliff, G. Jung, Robby, and G. Singh. Model-checking middleware-based event-driven real-time embedded software. In *Proceedings of the First International Symposium on Formal Methods for Components and Objects (FMCO 2002)*, November 2002.
8. X. Deng, M. B. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *Proceedings of the 24th International Conference on Software Engineering*. IEEE Press, 2002.
9. M. B. Dwyer, Robby, X. Deng, and J. Hatcliff. Space reductions for model checking quasi-cyclic systems. In *Proceedings of the Third International Conference on Embedded Software*, 2003.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Pub. Co., January 1995.
11. H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In *Proceedings of Eighth International SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, pages 217+. Springer-Verlag, 2001.
12. P. Godefroid. Model-checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 174–186, Jan. 1997.
13. J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. P. Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 25th International Conference on Software Engineering*. IEEE Press, May 2003.
14. T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In O. Grumberg, editor, *Computer-Aided Verification, 12th International Conference*, volume 1855, pages 20–35. Springer, 2000.
15. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
16. R. Iosif. Symmetry reduction criteria for software model checking. In *Proceedings of Ninth International SPIN Workshop*, volume 2318 of *Lecture Notes in Computer Science*, pages 22–41. Springer-Verlag, Apr. 2002.
17. M. Jones and E. Mercer. Explicit state model checking with hopper. In *Proceedings of Eleventh International SPIN Workshop*, volume 2989 of *Lecture Notes in Computer Science*, pages 146–150. Springer-Verlag, April 2004.
18. F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *Proceedings of Eighth International SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, pages 80–102. Springer-Verlag, 2001.
19. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003.

Models for Contract Conformance

Sriram K. Rajamani and Jakob Rehof

Microsoft Research
{sriram, rehof}@microsoft.com

Abstract. We have implemented a contract checker for asynchronous, message-passing applications to check that service implementations conform to behavioural contracts. Our contract checker is based on a process algebraic theory of conformance and is implemented on top of a software model checker, Zing. The purpose of this paper is to explain the model construction implemented by our contract checker and how it is related to a mathematical theory of conformance. In addition, we point out current and future research directions in model construction for conformance checking in the presence of channel-passing.

1 Introduction

Asynchronous message-passing programming is becoming increasingly important through the Internet. Web services hold the promise of making a new class of distributed applications possible that can automate business processes that are currently manual. Such applications are loosely coupled and may not be written by a single organization. There is an increasing interest in being able to specify the public behaviour of message-passing components by means of *contracts* in order to enable their composition without knowledge of their implementation.

We are interested in checking conformance between a message-passing component (a service) and a behavioural contract specifying the legal sequences of messages that can be received or sent by the component. We have implemented a conformance checker on top of our software model checker, Zing [3,2], and have used it to check contract conformance for applications written in high-level languages, including C# applications using messaging libraries of the .NET framework and applications written in experimental language extensions with message-passing primitives.

Our contract checker works by extracting models from source code into the language accepted by Zing. In addition to extracting Zing models from source code the checker must abstract models into a form that allows a mathematical notion of conformance to be soundly checked. This paper focuses on the latter problem. We describe the architecture of contract and service models and explain how this model construction is related to our theory of conformance. In addition, we point out current and future research directions in model construction for conformance checking in the presence of channel-passing.

T. Margaria and B. Steffen (Eds.): ISoLA 2004, LNCS 4313, pp. 181–196, 2006.

2 Contract Checker

2.1 Services and Contracts

In order to illustrate our approach, we will use a typical web service scenario as a running example, consisting of three distributed components – a client called Traveler, a TravelAgent service and an Airline service. The travel agent service is a client of the airline service. The system is outlined in Figure 1 which shows the types and directions of messages that can flow between the three components:

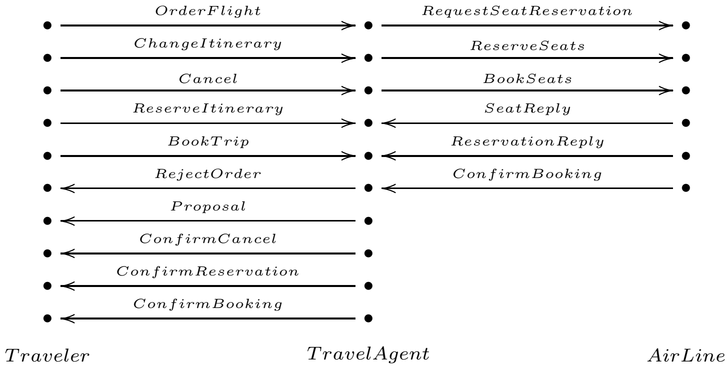


Fig. 1. Web service system

Such a system can be implemented in a conventional programming language, such as C# or Java, on top of a message-passing library. Alternatively, it can be implemented using special, web service oriented languages with built-in support for message-passing operations, as are emerging in new standards such as BPEL4WS (Business Process Execution Language for Web services), WSFL (Web Services Flow Language) and WSCL (Web Service Conversation Language).

There is an increasing interest in being able to specify the public behaviour of service components by means of *contracts*, in order to enable their composition without knowledge of their implementation. By specifying *temporal behaviour* – legal sequences of messages – contracts go beyond current standard specifications as found in, e.g., WSDL, which state non-behavioural properties, such as data formats and directions of exchanged messages. Contracts for our example services appear in Figure 5.

2.2 Zing Language and Models

The Zing model checker accepts an expressive input language, also called Zing [1], which includes concurrency and message-passing primitives. The sequential sublanguage of Zing is a subset of C# with no inheritance and fewer base types and supports classes as reference types with methods, dynamic object creation

async-call-statement :
`async` *invocation-expression*;

select-statement :
`select` *[select-qualifiers]*{*join-statements*}

join-statement :
join-list– >*embedded-statement*
`timeout`– >*embedded-statement*

join-pattern :
`wait`(*boolean-expression*)
`receive`(*expression*,*expression*)
`event`(*integer-expression*,
integer-expression,
boolean-expression)

send-statement :
`send`(*expression*,*expression*);

select-qualifier :
`end first visible`

join-list :
join-pattern
join-list&&*join-pattern*

event-statement :
`event`(*integer-expression*,
integer-expression,
boolean-expression)

Fig. 2. Zing concurrency and communication primitives

and structures. Zing supports fairly direct compilation from source languages, such as C# or Java, or special message-passing languages.

Figure 2 shows the core concurrency and communication primitives in Zing. The asynchronous call statement spawns a parallel process defined by a method. For example, `async o.foo(); S` creates a new instance of `foo` and runs it asynchronously in parallel with the continuation `S`. Zing automatically explores all interleavings between parallel processes. The statement `send(`c`,`e`)` sends the value denoted by e on the channel denoted by c. Zing channels are unbounded FIFO buffers and the send is asynchronous. The `select` statement is a powerful synchronization primitive, which blocks waiting for one of its join patterns to be satisfied. One of the satisfied patterns is chosen nondeterministically and the associated continuation, separated by `->`, is executed. The pattern `wait(`b`)` waits for boolean expression b to become true. The pattern `receive(`c`,`x`)` waits until the queue denoted by c becomes nonempty after which it consumes the value at the front of the queue and binds it to x. The pattern `event(`m`,`n`,`b`)` can always be triggered. Choosing this pattern creates a corresponding internal Zing event, $e(m, n, b)$, that can be observed and captured by the Zing runtime. The effect is the same when executing an `event` statement. Complex join patterns can be obtained with the operator `&&` which requires the constituent patterns to be satisfied in any order.

An internal nondeterministic choice between statements S_1 and S_2 can be expressed as `select`{ `wait(true) ->` S_1 `wait(true) ->` S_2}. In a select statement, the qualifier `visible` tells the Zing runtime to generate a special internal event, called τ, whenever a nondeterministic choice is made between patterns. This facility is used to observe the structure of nondeterminism of a model and is essential in our conformance checker (see Section 5). We bound queue sizes by coding a blocking send, as `select`{ `wait(sizeof(`c`) <= k) -> send(`c`,e);`}.

2.3 Tool Architecture

The architecture of our contract checker is shown in Figure 3. It implements a theory of conformance [11,10] on top of the Zing software model checker [3,2]. The conformance check is applied to one service implementation at a time, together with the contracts the service depends on.

A contract C can be in two distinct relations to a service S. First, C can specify the service – meaning that S must *implement* C – in which case we call C the *exported* contract of S. We assume that a service has exactly one exported contract. Second, a contract C can specify (i.e., be the exported contract of) some other service that S is a client of, in which case we call C an *imported* contract of S. We assume that S may have multiple imported contracts.

The inputs to the conformance checker are a service implementation, its exported contract, and the imported contracts it depends on. The service and its imported contracts are combined into a Zing model, called the implementation model. The exported contract is compiled into another Zing model, called the exported contract model. These two Zing models are compiled into executables (.dll files) which are fed to the Zing conformance checker. The checker decides

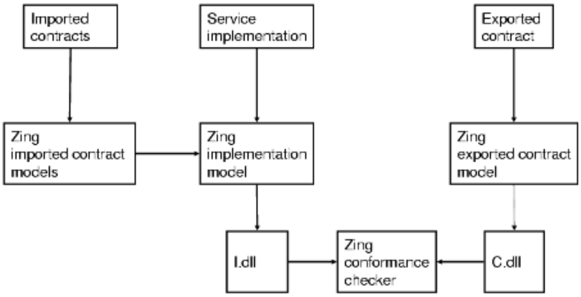


Fig. 3. Contract checking process

```
Contract TravelContract =
    OrderFlight? ->
       RejectOrder! Stop
     #
       Proposal!
       repeat
            ChangeItinerary?  ->
               RejectOrder! # Proposal!
          + Cancel?           ->
               ConfirmCancel! Stop
          + ReserveItinerary? ->
               RejectOrder!
             #
               ConfirmReservation!
               BookTrip? ->  ConfirmBooking!
        end
```

```
Contract AirLineContract =
   repeat
       RequestSeatReservation? -> SeatReply!
       ReserveSeats? -> ReservationReply!
       BookSeats? -> ConfirmBooking! Stop
   end
```

Fig. 4. Exported contract (left) and imported contract (right)

```
class TravelAgent: TravelContract{
Itin itin;
PO po;
CreditInfo creditinfo;

void Run(TravelAgentChannel Client){
 receive(Client,                     // (1)
         OrderFlight(order, itinerary)) ->
 {
  AirLineChannel AirLineService =    // (2)
         AirLine.Connect();

  if (! ValidateOrder()){
   send(Client, RejectOrder(po, itin));
   Stop;
  }
  else
   SendProposal(Client, AirLineService);
 }

 while (true){
  select
  {
   receive(Client, ChangeItinerary(po, itin))->
   {
    if (! ValidateOrder())
     send(Client, RejectOrder(po, itin));
    else
     SendProposal(Client, AirLineService);
   }
   receive(Client, Cancel(po, itin)) ->
   {
    Clear(po, itin);
    send(Client, ConfirmCancel(po, itin));
    break;
   }
   receive(Client,
           ReserveItinerary(po, creditinfo))->
   {
    send(AirLineService, ReserveSeats(sr)); // (3)
    receive(AirLineService,
            ReservationReply(reply))->
    {
     if (reply.success){
      send(Client,
           ConfirmReservation(po, itin));
      BookTrip();
      break;
     }
     else
      send(Client, RejectOrder(po, itin));
    } // receive
   } // receive
  } // select
 } // while
} // Run
}
```

Fig. 5. Service implementation (initial Zing model)

```
class TAmsg{
        static int OrderFlight = 0;
        static int RejectOrder = 1;
        static int Proposal = 2;
        static int ChangeItinerary = 3;
        static int Cancel = 4;
        static int ConfirmCancel = 5;
        static int ReserveItinerary = 6;
        static int ConfirmReservation = 7;
        static int BookTrip = 8;
        static int ConfirmBooking = 9;
};


class TravelAgentModel{
  static activate void Run(){

    // (1): Actions on exported contract channel
    // are abstracted as events
    //
    event(TAmsg.OrderFlight,0,false);

    // (2): Imported contract is substituted for service
    //
    Imp_AirLineChannel AirLineService =
                      new_Imp_AirLineContract();
    select visible{
       wait(true) -> {
                        event(TAmsg.RejectOrder, 0, true);
                        goto Stop;
                     }
       wait(true) ->    SendProposal();
    }
    while (true){
       select visible
       {
             event(TAmsg.ChangeItinerary, 0, false)->
             {
               select visible{
                 wait(true) ->
                      event(TAmsg.RejectOrder, 0, true);
                 wait(true) ->
                      SendProposal();
               }
             }
             event(TAmsg.Cancel, 0, false)->
             {
               event(TAmsg.ConfirmCancel, 0, true);
               goto exit;
             }
             event(TAmsg.ReserveItinerary, 0, false)->
             {
               // (3): Actions on imported contract channels
               // are concrete sends and receives
               //
               send(AirLineService, ALmsg.ReserveSeats);
               receive(AirLineService, ALmsg.ReservationReply)->
               select visible
               {
                 wait(true) -> {
                   event(TAmsg.ConfirmReservation, 0, true);
                   BookTrip();
                   goto exit;
                 }
                 wait(true) ->
                      event(TAmsg.RejectOrder, 0, false);
               }
             }
       }
    }
    exit:
    Stop:
  }
}
```

Fig. 6. Zing implementation model (abstract model)

whether the implementation model conforms to the exported contract model by exploring the state spaces of the models.

An important task in model construction is abstraction, whereby parts of the source program that are irrelevant for the properties we are interested in are thrown away, see e.g. [8,12]. We find it useful to separate the task of model compilation from the task of abstraction, since it allows developers familiar with compiler technology but less familiar with advanced program analysis to produce initial Zing models. The Zing model checking framework then performs abstraction and other model transformations on the initial models. Since this paper focuses on the specific model construction used in contract checking, we will assume for simplicity that we are given an initial Zing model as our service implementation.

Figure 5 contains excerpts of an example initial Zing model [1] for the TravelAgent service outlined in Figure 1. The model is defined by class `TravelAgent` together with definitions of the exported and imported contracts it depends on. The `TravelAgent` class is annotated with the name of a contract, `TravelContract`, indicating the exported contract of the service. The contract definition is shown in Figure 4, left. The `TravelAgent` service is a client of another service, `AirLineService`. Its contract definition is shown in Figure 4, right. This contract is imported by the `TravelAgent` service.

A contract specifies the externally visible, legal sequences of message types accepted by the service, abstracting away all internal computation of the service as well as message payloads. Contracts are defined in a succinct CSP-like notation [13], where `T?` denotes receiving a message of type `T`, `T!` denotes sending a message of type `T`, $P\#Q$ denotes the internal choice between P and Q, and $P{+}Q$ denotes external choice.

3 Principles of Contract Checking

3.1 Compositional Conformance Checking

A central idea in our contract checker is to check conformance between services and contracts *compositionally.* Compositionality means that only one service implementation is checked against its contract at a time, by *substituting* contracts for the other services, if any, that the service uses. Because contracts abstract from the internal computation of the services they specify, substituting contracts for services enhances model checking performance. It also allows us to check a service without relying on implementation code for other services that are used. This is important for web service applications, because different services may be developed by different organizations, and the code for other services may not be available.

Contracts must be substitutable for services that conform to them. The exact meaning of substitutability depends on the properties contract checking should

[1] The example has been edited for illustrative purposes and to keep it within bounds, but it is representative of models that have been generated by our tool on real applications.

guarantee. Our contract checker guarantees *stuck-freedom* of a system of services. A stuck state is one in which a component is blocked waiting to receive a message that is not sent (deadlock), or a component sends a message that is not received (unreceived messages). Stuck-freedom, then, ensures that a message exchange proceeds with consistent expectations between senders and receivers about which messages are sent and when. The Zing conformance checker implements a theory of *stuck-free conformance* [11,10] that we have developed for CCS [18,19]. Substitutability can be formalized as the following property, where P is a service, Q is any client of P, C is the contract of P, $\leq$ denotes conformance, and $\mid$ denotes parallel composition:

- If $P \leq C$, then $C \mid Q$ stuck-free implies $P \mid Q$ stuck-free, for all Q.

When checking stuck-freedom against any client Q, it is therefore *safe* to substitute C for P. Our conformance relation $\leq$ described in Section 3.2 guarantees substitutability.

3.2 Stuck-Free Conformance

Our stuck-free conformance [11,10] relation between CCS processes [18,19] is the largest relation $\leq$ such that, whenever $P \leq Q$, then the following conditions hold:

C1. If $P \xrightarrow{\tau^*\lambda} P'$ then there exists Q' such that $Q \xrightarrow{\tau^*\lambda} Q'$ and $P' \leq Q'$.
C2. If P can refuse X while ready on Y, then Q can refuse X while ready on Y.

Here, $P \xrightarrow{\tau^*\lambda} P'$ means that P can transition to P' on a sequence of hidden actions, τ, and a visible action, λ. A process is called *stable*, if it cannot do any τ-actions. If X and Y are sets of visible actions, we say that P *can refuse* X *while ready on* Y, if there exists a stable P' such that $P \xrightarrow{\tau^*} P'$ and (i) P' *refuses* X, i.e., P' cannot do a co-action of any action in X, and (ii) P' *is ready on* Y, i.e., P' can do every action in Y. In condition [C2] above, the ready sets Y range only over singleton sets or the empty set. The restriction to singleton or empty ready sets is important since it allows specifications to be more nondeterministic than implementations. The references [11,10] contain a detailed explanation of this and other aspects of stuck-free conformance. Zing's conformance checking algorithm implements the relation $\leq$ and is described in [3].

We can show [11,10] that conformance $\leq$ is a simulation relation satisfying the following two properties:

1. (Precongruence) $P \leq Q$ implies $\mathscr{C}[P] \leq \mathscr{C}[Q]$ for all CCS contexts $\mathscr{C}$
2. (Preservation) If $P \leq Q$ then P not stuck-free implies Q not stuck-free

Precongruence says that stuck-free conformance is preserved by all CCS contexts. Preservation says that stuck-free conformance preserves the ability to get stuck. Together, the two properties immediately imply:

3. (Substitutability) If $P \leq Q$ then $\mathscr{C}[Q]$ stuck-free implies $\mathscr{C}[P]$ stuck-free, for all contexts $\mathscr{C}$

3.3 Exported and Imported Contracts

We think abstractly of a service P that exports a contract C as a CCS process with a free channel name a on which it implements its exported contract and with a set of channel names $b_1, \ldots, b_n$ connecting it to other services $S_1, \ldots, S_n$ that it is a client of. The whole system can then be thought of as:

$$S_P(a) = (\texttt{new}\ b_1 \ldots b_n)(S_1(b_1) \mid \ldots \mid S_n(b_n) \mid P(a, b_1, \ldots, b_n))$$

The new-binder on the names $b_1, \ldots, b_n$ indicates that we think of the connections to the other services $S_1, \ldots, S_n$ as being local, i.e., no other processes can interact with P on any of these names.

The tasks in contract checking P for stuck-freedom are:

1. Check that $S_P(a) \leq C(a)$, where C is the exported contract of P
2. Check that P does not get stuck with any of the services $S_1, \ldots, S_n$ it is a client of

To enable compositional checking we employ the substitutability principle in model construction:

- Imported contracts are substituted for the services they specify

Hence, our contract checker constructs the model of the system:

$$S_P(a)^M = (\texttt{new}\ b_1 \ldots b_n)(C_1(b_1) \mid \ldots \mid C_n(b_n) \mid P(a, b_1, \ldots, b_n)^M)$$

where the contracts $C_1, \ldots, C_n$ have been used instead of the services they specify (Q^M denotes a model of Q.)

The exported and imported roles of a contract require different semantics in contract checking. Because conformance with exported contracts (task 1) must ensure stuck-free substitutability in *all* possible contexts, the conformance relation $\leq$ is formulated in terms of Milner's *commitment* semantics of processes [18,19]. Commitments are generated by communication actions regardless of the environment and allows reasoning over an open system in all contexts. On the other hand, stuck-freedom against a particular process (task 2) is a property of a closed system. Hence, we interpret a contract occurring in its imported role as a concretely executing process with *reaction* semantics [18,19], where only internal reactions (τ actions) generated from an action and its co-action are possible.

4 Model Abstraction

Constructing an abstract service model from a given initial model involves:

1. Omit data and statements that are irrelevant for conformance checking
2. Abstract actions on the message channels
3. Construct suitable models of imported contracts and integrate them into the service implementation model

Figure 6 shows the abstract service implementation model generated from the initial model. Data other than communication channels and message types are omitted, and the control structure of the initial model is abstracted. For example, conditionals are transformed into nondeterministic choices.

4.1 Implementation Model

As mentioned in Section 3.3, imported contracts are interpreted under *reaction* semantics, whereas exported contracts are interpreted under *commitment* semantics. The `event` statement is used in Zing to model commitments, whereas the `send` and `receive` statements are concrete communication statements that operate on message queues and are used to model actions under reaction semantics. Referring to our example, actions on the `Client` channel are compiled into Zing `event` statements, whereas actions on the `AirLineService` channel are abstracted into `send` and `receive` statements. To see how that manifests in the abstracted model, compare the statements with comments numbered `(1)` and `(3)` in Figure 6 with the corresponding statements in the initial program (Figure 5).

The Zing conformance checker can run two models (given to it as distinct dll's) and compare events generated by the models. We use the numerical arguments to encode event numbers (denoting types of events in this application) and channel numbers, and we use the boolean argument to indicate the direction of the message. Hence, we interpret `event(m,n,true)` as standing for "a message of type `m` is sent on channel `n`".

Substitution of imported contracts for the services they specify is illustrated in the statement with comment numbered `(2)` in Figure 6. Here, the connection to the `AirLineService` is abstracted by calling the constructor method of the class `Imp_AirLineContract`, which is the model of the imported contract (discussed further in Section 4.2). A side-effect of calling this constructor is to fork off a concurrent process, compiled from the `AirLineContract` specification in Figure 5 (bottom right), which implements the semantics of the imported contract. That way, when the implementation connects to a service whose contract it imports, the model will start a process that implements its *contract*, and the service model will start interacting with the contract process through message queues. Rather than transmitting real messages, the model transmits numbers indicating the types of messages, according a numerical coding of message types as shown in Figure 6 (top).

4.2 Contract Models

The exported contract model of the `TravelContract` defined in Figure 5 is shown in Figure 7, left. It represents all actions as commitments (`event`'s). The conformance check `TravelAgentModel` $\leq$ `Exp_TravelContract` is performed by compiling two distinct dll's from the service implementation and the exported contract model. The `Run` methods of both of these models are marked as `static activate`, which tells Zing to treat these methods as start-up ("main") processes. After launching the processes, the Zing conformance checker drives them through their state spaces, collects the events generated and compares them according to the mathematical definition of stuck-free conformance [11,10].

The imported contract model of the `AirLineContract` defined in Figure 5 is shown in Figure 7, right. The constructor method, `new_Imp_AirLineContract`, spawns a new instance of the `Run` method of the model with a fresh channel

```
class Exp_TravelContract{

static activate void Run(){
 event(TAmsg.OrderFlight,0,false);
 select visible
 {
   wait(true) ->
   {
    event(TAmsg.RejectOrder,0,true);
    goto Stop;
   }
   wait(true) ->
   {
    event(TAmsg.Proposal,0,true);
    while(true)
    {
     select visible
     {
      event(TAmsg.ChangeItinerary,0,false)->
      {
       select visible
       {
        wait(true) ->
         event(TAmsg.RejectOrder,0,true);
        wait(true) ->
         event(TAmsg.Proposal,0,true);
       }
      }
      ...
     }
    }
   }
 }
 Stop:
}
};
```

```
class Imp_AirLineContract{

static Imp_AirLineChannel
       new_Imp_AirLineContract()
{
  Imp_AirLineContract AL =
      new Imp_AirLineContract;
  Imp_AirLineChannel c =
      new AirLineChannel;
  Client = c;
  async AL.Run(c);
  return c;
}

static void Run(AirLineChannel c){
  while(true){
   select{
    receive(c, ALmsg.RequestSeatReservation)->
               send(c, ALmsg.SeatReply);
    receive(c, ALmsg.ReserveSeats) ->
               send(c, ALmsg.ReservationReply);
    receive(c, ALmsg.BookSeats) ->
               send(c, ALmsg.ConfirmBooking);
   }
  }
}
};
```

Fig. 7. Zing exported contract model (left) and Zing imported contract model (right)

of type `Imp_AirLineChannel` as argument. The `async` keyword in Zing means that the called method is run as a concurrent process. This abstraction scheme allows us to model multiple independent instantiations of a service by calling the corresponding contract constructor each time. The `Run` method contains the model code derived from the contract definition, by interpreting its actions as `send` and `receive` actions on message channels carrying type codes. When called from the service implementation model, the imported contract model fires up as a concurrent process and interacts with it.

5 Stability and Message Queues

In this section we consider two important aspects of how the theory of stuck-free conformance [11,10] supports the model construction described above. We consider in more detail the role of nondeterminism and hidden (τ) events, and we discuss the problem of modeling ordered message queues and type-dependent messaging constructs.

5.1 Stability

A process is called stable, if it cannot generate a τ-action. Such actions model internal, hidden computation steps. During conformance checking, Zing's runtime generates events when statements are executed. By observing these events, the conformance checker implements the definition of stuck-free conformance given above. Events that are generated from the Zing `event` statement are classified as external events, all other events are classified as τ-events by the conformance checker. Such other events include the call of a method, the execution of an assignment statement, a concrete reaction generated by a communication (message being placed into or removed from a message queue), forking a process, etc. Zing can therefore observe whether a state is stable or not and the external actions, if any, that can be generated from it. Even though Zing models can become very complicated due to the use of expressive language features, we can establish a very direct relation between our Zing models and the theory of conformance for CCS, since both systems are labeled transition systems.

Instability is closely related to internal, hidden nondeterminism. Contracts may be more nondeterminstic than the implementations they specify. An internal non-deterministic choice between statements S_1 and S_2 can be expressed in Zing as `select visible{ wait(true) ->` S_1 `wait(true) ->` S_2`}`. This statement generates a τ-event in the Zing runtime and corresponds directly to the process algebraic encoding of internal nondeterministic choice $S_1 \# S_2$ as $\tau.S_1 + \tau.S_2$, where + denotes external choice. Subtle cases can arise in situations with mixed choice, of the form $a.P + \tau.Q$. Notice that $P = a + \tau.b$ is not equivalent to $Q = a \# b$ (we have $P \leq Q$ but *not* $Q \leq P$), and even $a \leq a + \tau$ is false, as can be verified from the definition. Such issues can arise in model construction. For example, consider a program that receives messages by an external choice over message types with a timeout case attached:

```
select{
        receive(c, T1(...)) -> $S_1$
        ...
        receive(c, Tn(...)) -> $S_n$
        Timeout -> $S_{n+1}$ }
```

In abstracting this statement to commitment actions for conformance checking with an exported contract, we generate a statement of the form

```
select visible{
            receive(MsgType.T1, 0, false) -> $S_1^M$
            ...
            receive(MsgType.Tn, 0, false) -> $S_n^M$
            wait(true) -> $S_{n+1}^M$ }
```

The latter statement is equivalent to the CCS expression `T1.`S_1^M `+ ... + Tn.`S_n^M `+` τ`.`S_{n+1}^M.

5.2 Ordered Message Queues

Asynchronous message-passing via buffered, bidirectional and ordered communication channels is essential in practical applications. Two processes communicating with each other via a bidirectional ordered channel maintain distinct orderings on messages travelling in each direction, so two queues are needed to implement such a channel, one for the communication in each direction. Zing supports ordered message queues, and we use pairs of such to implement bidirectional channels. However, since our `event` based models of actions on channels with exported contracts abstract away from queues, it requires an argument to show that the conformance check remains sound in the presence of a particular, ordered queue semantics. Moreover, the ability to specify type-dependent receives, as exemplified by the form `select{ receive(c, T1(...)) -> ...}`, is very convenient for many programming tasks. The statement blocks until a message of specified type `T1 ...` arrives at the front of the input queue associated with channel `c`. The type-dependent nature of receive statements raises a similar question about soundness of conformance checking.

These questions are best approached rigorously by considering an unbounded queue encoding in CCS (see [19]). We will define two queues, a left queue Q^L and a right queue Q^R. Let $\mathbb{A}$ be a finite alphabet of message types, let A range over $\mathbb{A}$, and let w range over $\mathbb{A}^*$. We define the queue Q^L as follows:

$$Q^L_{\langle\rangle} \quad = \textstyle\sum_A(\mathtt{in}^L_A\mathtt{?}.Q^L_{\langle A\rangle} + \mathtt{empty}^L\mathtt{!}.Q^L_{\langle\rangle})$$

$$Q^L_{\langle w,A'\rangle} = \textstyle\sum_A(\mathtt{in}^L_A\mathtt{?}.Q^L_{\langle A,w,A'\rangle} + \mathtt{out}^L_{A'}\mathtt{!}.Q^L_{\langle w\rangle})$$

We have used the encoding $\mathtt{in}^L_A\mathtt{?}$ for the action of receiving a message of type A on queue Q^L, and similarly for sending, $\mathtt{out}^L_A\mathtt{!}$, so we can express message order based on message types in $\mathbb{A}$ as well as type-dependent receives from these queues. The queue Q^R is defined analogously, using names $\mathtt{in}^R_B$, $\mathtt{out}^R_B$ with types B drawn from a finite set $\mathbb{B}$. Now consider two processes, say C and P, communicating via a bidirectional queue, with messages incoming to C and outgoing from P typed from $\mathbb{A}$, and messages incoming to P and outgoing from C typed in $\mathbb{B}$. C could be an imported contract model and P an implementation model that uses it. We can write the system as $C' \mid P'$ with

$$C' = (\mathtt{new}\ \mathtt{in}^L_{A_1} \ldots \mathtt{in}^L_{A_n})(C \mid Q^L_{\langle\rangle})$$

and

$$P' = (\mathtt{new}\ \mathtt{in}^R_{B_1} \ldots \mathtt{in}^R_{B_m})(P \mid Q^R_{\langle\rangle})$$

Notice that the free names in C' are of the form $\mathtt{out}^L_A$ or $\mathtt{empty}^L$, and the free names in P' are of the form $\mathtt{out}^R_B$ or $\mathtt{empty}^R$. In C, sending a message of type A is coded as the action $\mathtt{in}_A\mathtt{!}$ and receiving a message of type B is coded as the action $\mathtt{out}^R_B\mathtt{?}$. Similarly, in P, sending a message of type B is coded as the action $\mathtt{in}^R_B\mathtt{!}$, and receiving a message of type A is coded as the action $\mathtt{out}^L_A\mathtt{?}$.

To see that our model construction remains sound in the presence of these queues and codings, observe that the system $C' \mid P'$ can be written as $\mathscr{C}^P_Q[C]$, where $\mathscr{C}^P_Q$ is the context

$$\mathscr{C}^P_Q = (\mathtt{new}\ \mathtt{in}^L_{A_1} \ldots \mathtt{in}^L_{A_n})([] \mid Q^L_{\langle\rangle}) \mid P'$$

Hence, if we know that $R \leq C$, we also know that $\mathscr{C}^P_Q[R] \leq \mathscr{C}^P_Q[C]$, for all P, by the Precongruence property of conformance. It follows that conformance is preserved by the type-dependent ordered queue semantics, and therefore it is sound to check conformance in abstraction from the queue.

The previous argument shows soundness, but it does not rule out that the nature of conformance can be impacted by the presence of a queue. For example, consider the process

$$X = \mathtt{in}^R_{A_1}\mathtt{!} \mid \mathtt{in}^R_{A_2}\mathtt{!}$$

which sends messages A_1 and A_2 to the same queue in parallel. Compare with the process

$$Y = (\mathtt{in}^R_{A_1}\mathtt{!.in}^R_{A_2}\mathtt{!}) \# (\mathtt{in}^R_{A_2}\mathtt{!.in}^R_{A_1}\mathtt{!})$$

By the definition of conformance, we have $X \leq Y$, but *not* $Y \leq X$. However, we do have $\mathscr{C}^P_Q[Y] \leq \mathscr{C}^P_Q[X]$, for all P. The reason is that the act of placing a message into the queue generates a τ action that signifies the determination of a particular ordering of messages. An consequence of this situation is that our conformance check does not determine that, say,

```
select visible {wait(true) -> {send(c,A₁());send(c,A₂())}
                wait(true) -> {send(c,A₂());send(c,A₁())}}
```

can be a valid implementation of the specification X in this context (we have shown the implementation as it could look prior to `event` abstraction.) Hence, our checks may be overly conservative in some cases. The problem of adjusting the theory of conformance to a particular queue semantics may be an interesting one, but we have not pursued it.

6 Current and Future Work

In this section we discuss current work on tool integration and two generalizations of our method of model construction that we are currently investigating.

Tool Integration. Our contract checker is being integrated into a development environment for a language with web service oriented, message-passing extensions. The contract checker is being integrated into the compilation pipeline, much like a type checker, where error messages arising from contract conformance violations are reflected back to the source code automatically. Model abstraction is performed automatically from the source and is currently restricted to the core message-passing sublanguage. Extensions to the abstraction to handle more language constructs are foreseen.

Channel Passing. Channel passing requires us to generalize our methods from CCS to the π-calculus [19]. Inspired by [15] we have proposed a behavioural type system for the π-calculus that is based on model extraction into CCS models and simulation as subtyping [7]. Since stuck-free conformance is a restriction of CCS simulation, it is sound to use stuck-free conformance in that system. However, the system has limitations it would be interesting to lift. In particular, the system is asymmetric in that sending a channel is correlated with importing its contract and receiving a channel is correlated with exporting its contract. While this is natural in many important practical scenarios, it may not be general enough, and it is logically less satisfying. We are currently working towards lifting this restriction.

Multiple Exported Contracts. Our model construction assumes that a service implementation has at most one exported contract. But a service $P(a, b)$ could serve one client on channel a and another on b. With channel-passing this is an important issue. One solution is to specify the actions on a and b as a single contract $C(a, b)$. The problem is where to put the specification and where to initiate the contract checks. If we specify two contracts, $C_1(a)$ and $C_2(b)$, the problem is how they are related. If we assume they are independent, then the service must implement the contract $C_1(a) \mid C_2(b)$ which imposes many restrictions on the implementation. We could define a projection operator, $\downarrow$, such that (i) $P \leq (P \downarrow_a) \mid (P \downarrow_b)$ and (ii) $(P \downarrow_a) \leq C_1(a)$ and $(P \downarrow_b) \leq C_2(b)$. The natural operator $\downarrow_a$ is hiding [22], turning actions other than a into τ events. This operator does not automatically satisfy (i). For example,

$$(a + b) \not\leq (a + \tau) \mid (\tau + b) = ((a + b) \downarrow_a) \mid ((a + b) \downarrow_b)$$

The failure of conformance here is due to the readiness constraint in condition [C2] in the defintion of conformance. We can consider restricting the readiness condition in the definition of conformance to encompass only output actions, in order to validate more general projections (doing so is theoretically possible). We can require clients of the contracts to be "isolated", so that they only use contracts independently. We are currently working out the latter two ideas in more detail.

7 Related Work

Stuck-free conformance builds on both the CSP tradition [13,22] and the CCS tradition [18,19]. It is closely related to and inspired by stable failures refinement in CSP [6,13,22] and the refinement checker FDR [22]. Stuck-free conformance is also related to the refusals preorder for CCS as developed in Iain Phillips' theory of refusal testing [20,5] and to 2/3 bisimulation by Larsen and Skou [17]. Our papers on stuck-free conformance [11,10] contain more in-depth comparisons.

We are not aware of a tool for source-level compositional conformance analysis of message-passing programs whose architecture of models is as described in this paper.

Many related tools based on model checking have been developed, including Spin [14] and dspin [16]. Model checkers have also been used to check Java programs either directly [24,23,21] or by constructing slices or other abstractions [9]. The slam project [4] has similar goals to zing and has been very successful in checking control-dominated properties of device drivers written in C. Unlike zing, it does not handle concurrent programs, and it is unable to prove interesting properties on heap-intensive programs.

References

1. Zing Language Specification – http://research.microsoft.com/zing.
2. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *CAV 2004:16th International Conference on Computer Aided Verification, Boston, Massachusetts, July 2004*, LNCS. Springer-Verlag, 2004.
3. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: Exploiting program structure for model checking concurrent software. In *CONCUR 2004: Fifteenth International Conference on Concurrency Theory, London, U.K., September 2004*, LNCS. Springer-Verlag, 2004. Invited paper.
4. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, January 2002.
5. E. Brinksma, L. Heerink, and J. Tretmans. Developments in testing transition systems. In *Testing of Communicating Systems*, IFIP TC6 10th International Workshop on Testing of Communicating Systems, pages 143 – 166. Chapman & Hall, 1997.
6. S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.
7. S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: Model checking message-passing programs. In *POPL 02: ACM Principles of Programming Languages.* ACM, 2002.
8. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE 00: International Conference on Software Engineering*, pages 439–448. ACM, 2000.
9. M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *ICSE 01: International Conference on Software Engineering*, pages 177–187. ACM, 2001.
10. C. Fournet, C. A. R. Hoare, S. K. Rajamani, and J. Rehof. Stuck-free conformance theory for CCS. Technical Report MSR-TR-2004-09, Microsoft Research, 2004.
11. C. Fournet, C.A.R. Hoare, S.K. Rajamani, and J. Rehof. Stuck-free conformance. In *CAV 04: Computer-Aided Verification*, LNCS. Springer-Verlag, July 2004.
12. J. Hatcliff and M. Dwyer. Using the bandera toolset to model check properties of concurrent java software. In *CONCUR 2001*, LNCS 2154. Springer-Verlag, 2001.
13. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
14. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
15. A. Igarashi and N. Kobayashi. A generic type system for the Pi-calculus. In *POPL 01: Principles of Programming Languages*, pages 128–141. ACM, 2001.
16. R. Iosif and R. Sisto. dSPIN: A dynamic extension of SPIN. In *SPIN 99: SPIN Workshop*, LNCS 1680, pages 261–276. Springer-Verlag, 1999.
17. K.G. Larsen and A. Skou. Bisimulation through probabilistic testing. In *POPL 89: ACM Principles of Programming Languages*, pages 344–352. ACM, 1989.
18. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
19. R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.
20. I. Phillips. Refusal testing. *Theoretical Computer Science*, 50(2):241 – 284, 1987.

21. Robby, M. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *FSE 03: Foundations of Software Engineering*, pages 267–276. ACM, 2003.
22. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.
23. S. D. Stoller. Model-checking multi-threaded distributed Java programs. *International Journal on Software Tools for Technology Transfer*, 4(1):71–91, oct 2002.
24. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ASE 00: Automated Software Engineering*, pages 3–12, 2000.

Author Index