Luciano Baresi
Reiko Heckel (Eds.)

LNCS 3922

Fundamental Approaches to Software Engineering

9th International Conference, FASE 2006
Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006
Vienna, Austria, March 2006, Proceedings

Springer

Lecture Notes in Computer Science 3922

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

Editorial Board

Luciano Baresi Reiko Heckel (Eds.)

Fundamental Approaches to Software Engineering

9th International Conference, FASE 2006
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2006
Vienna, Austria, March 27-28, 2006
Proceedings

Volume Editors

Luciano Baresi
Politecnico di Milano, Dipartimento di Elettronica e Informazione
piazza Leonardo da Vinci, 32 - 20133 Milano, Italy
E-mail: baresi@elet.polimi.it

Reiko Heckel
University of Leicester, Department of Computer Science
University Road, LE1 7RH Leicester, UK
E-mail: reiko@mcs.le.ac.uk

Library of Congress Control Number: 2006922237

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

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

ISSN 0302-9743
ISBN-10 3-540-33093-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-33093-6 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11693017 06/3142 5 4 3 2 1 0

Foreword

ETAPS 2006 was the ninth instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference that was established in 1998 by combining a number of existing and new conferences. This year it comprised five conferences (CC, ESOP, FASE, FOSSACS, TACAS), 18 satellite workshops (ACCAT, AVIS, CMCS, COCV, DCC, EAAI, FESCA, FRCSS, GT-VMT, LDTA, MBT, QAPL, SC, SLAP, SPIN, TERMGRAPH, WITS and WRLA), two tutorials, and seven invited lectures (not including those that were specific to the satellite events). We received over 550 submissions to the five conferences this year, giving an overall acceptance rate of 23%, with acceptance rates below 30% for each conference. Congratulations to all the authors who made it to the final programme! I hope that most of the other authors still found a way of participating in this exciting event and I hope you will continue submitting.

The events that comprise ETAPS address various aspects of the system development process, including specification, design, implementation, analysis and improvement. The languages, methodologies and tools which support these activities are all well within its scope. Different blends of theory and practice are represented, with an inclination towards theory with a practical motivation on the one hand and soundly based practice on the other. Many of the issues involved in software design apply to systems in general, including hardware systems, and the emphasis on software is not intended to be exclusive.

ETAPS is a loose confederation in which each event retains its own identity, with a separate Program Committee and proceedings. Its format is open-ended, allowing it to grow and evolve as time goes by. Contributed talks and system demonstrations are in synchronized parallel sessions, with invited lectures in plenary sessions. Two of the invited lectures are reserved for "unifying" talks on topics of interest to the whole range of ETAPS attendees. The aim of cramming all this activity into a single one-week meeting is to create a strong magnet for academic and industrial researchers working on topics within its scope, giving them the opportunity to learn about research in related areas, and thereby to foster new and existing links between work in areas that were formerly addressed in separate meetings.

ETAPS 2006 was organized by the Vienna University of Technology, in cooperation with:

- European Association for Theoretical Computer Science (EATCS);
- European Association for Programming Languages and Systems (EAPLS);
- European Association of Software Science and Technology (EASST);
- Institute for Computer Languages, Vienna;
- Austrian Computing Society;
- The *Bürgermeister der Bundeshauptstadt Wien*;
- Vienna Convention Bureau;
- Intel.

The organizing team comprised:

Chair:	Jens Knoop
Local Arrangements:	Anton Ertl
Publicity:	Joost-Pieter Katoen
Satellite Events:	Andreas Krall
Industrial Liaison:	Eva Kühn
Liaison with City of Vienna:	Ulrich Neumerkel
Tutorials Chair, Website:	Franz Puntigam
Website:	Fabian Schmied
Local Organization, Workshops Proceedings:	Markus Schordan

Overall planning for ETAPS conferences is the responsibility of its Steering Committee, whose current membership is:

Perdita Stevens (Edinburgh, Chair), Luca Aceto (Aalborg and Reykjavík), Rastislav Bodík (Berkeley), Maura Cerioli (Genova), Matt Dwyer (Nebraska), Hartmut Ehrig (Berlin), José Fiadeiro (Leicester), Marie-Claude Gaudel (Paris), Roberto Gorrieri (Bologna), Reiko Heckel (Leicester), Michael Huth (London), Joost-Pieter Katoen (Aachen), Paul Klint (Amsterdam), Jens Knoop (Vienna), Shriram Krishnamurthi (Brown), Kim Larsen (Aalborg), Tiziana Margaria (Göttingen), Ugo Montanari (Pisa), Rocco de Nicola (Florence), Hanne Riis Nielson (Copenhagen), Jens Palsberg (UCLA), Mooly Sagiv (Tel-Aviv), João Saraiva (Minho), Don Sannella (Edinburgh), Vladimiro Sassone (Southampton), Helmut Seidl (Munich), Peter Sestoft (Copenhagen), Andreas Zeller (Saarbrücken).

I would like to express my sincere gratitude to all of these people and organizations, the Program Committee chairs and PC members of the ETAPS conferences, the organizers of the satellite events, the speakers themselves, the many reviewers, and Springer for agreeing to publish the ETAPS proceedings. Finally, I would like to thank the Organizing Chair of ETAPS 2006, Jens Knoop, for arranging for us to have ETAPS in the beautiful city of Vienna.

Edinburgh, January 2006

Perdita Stevens
ETAPS Steering Committee Chair

Preface

Software engineering aims to create a feedback cycle between academia and industry, proposing new solutions and identifying those that "work" in practical contexts. The conference on Fundamental Approaches to Software Engineering (FASE) —as one of the European Joint Conferences on Theory and Practice of Software (ETAPS)— is committed to this aim.

With the society increasingly relying on software, the ability to produce low-cost and high-quality software systems is crucial to technological and social progress. FASE provides software engineers with a forum for discussing theories, languages, methods, and tools derived from the interaction of academic research and real-world experience.

Contributions were sought targeting problems of practical relevance through fundamental contributions, based on solid mathematical or conceptual foundations, which could lead to improved engineering practices.

The response of the scientific community was overwhelming, with record submission numbers of 166 research papers and 7 tool papers. From these, 27 research papers and 2 tool papers were selected for publication, with an overall acceptance rate of 17%. The international character of the conference is underlined by the fact that just about one third of the authors are from European countries, while the others come from North America, Asia and Australia.

Accepted papers address topics like distributed and service-oriented computing, measurement and empirical software engineering, methods and tools for software development, validation and verification, model-based development, and software evolution. The scientific programme is complemented by the invited lecture of Francisco Curbera on "A Programming Model for Service Oriented Applications" and of Carlo Ghezzi on "Software Engineering: Emerging Goals and Lasting Problems".

We are deeply indebted to the 24 members of the Program Committee and the 123 additional reviewers for their invaluable time, spent reading and discussing a large number of papers and producing more then 500 reviews.

FASE 2006 was held in Vienna (Austria), hosted and organized by the Institute for Computer Languages at the Vienna University of Technology. Next year FASE will take place in Braga (Portugal). Being part of ETAPS, FASE shares the sponsoring and support acknowledged in the foreword. Heartfelt thanks are also due to Perdita Stevens for excellent and efficient global coordination and to Jens Knoop and his staff for their wonderful job as local organizers.

Finally, special thanks to all contributors and participants who, at the end of the day, are what this is all about.

Milano and Leicester, January 2006

Luciano Baresi
Reiko Heckel

Organization

Program Committee

Jan Øyvind Aagedal (SINTEF, Oslo, Norway)
Luciano Baresi (Politecnico di Milano, Italy) Co-chair
Jean Bézivin (University of Nantes, France)
Victor Braberman (University of Buenos Aires, Argentina)
Maura Cerioli (Università di Genova, Italy)
Matt Dwyer (University of Nebraska, USA)
Anthony Finkelstein (University College London, UK)
Harald Gall (University of Zurich, Switzerland)
Alan Hartman (IBM, Israel)
Reiko Heckel (University of Leicester, UK) Co-chair
Mehdi Jazayeri (University of Lugano, Switzerland and University of Vienna, Austria)
Antónia Lopes (University of Lisbon, Portugal)
Sandro Morasca (Università dell'Insubria, Italy)
András Pataricza (Budapest University of Technology and Economics, Hungary)
Mauro Pezzè (Università di Milano-Bicocca, Italy)
Arend Rensink (University of Twente, The Netherlands)
Leila Ribeiro (Federal University of Rio Grande do Sul, Brazil)
Andy Schürr (University of Darmstadt, Germany)
Gabi Taentzer (Technische Universität Berlin, Germany)
Tetsuo Tamai (University of Tokyo, Japan)
Sebastian Uchitel (Imperial College, UK)
Heike Wehrheim (University of Paderborn, Germany)
Michel Wermelinger (Open University, UK)
Alexander Wolf (University of Lugano, Switzerland and University of Colorado, USA)
Michal Young (University of Oregon, USA)

Referees

N. Aguirre
F. Asteasuain
L. Balmelli
A. Balogh
L. Barroca
A. Benczur
B. Braatz
C. Braghin
I. Brueckner
A. Capiluppi
W. Cazzola
A. Cherchago
J. Cook
S. Costa
E. Cota
I. Craggs
G. Csertán
J. de Lara
H. Dierks
G. Dodero
F. Dotti
S. Dustdar
K. Ehrig
C. Eisner
C. Ermel
J. Faber
E. Farchi
J. Fernandez-Ramil
M. Fischer
A. Fiuri

B. Fluri
L. Foss
M. Frias
D. Garbervetsky
H. Giese
L. Gönczy
R. Grønmo
T. Gyimóthy
J. Hall
J.H. Hausmann
J. Henkel
F. Herrmann
S. Holland
M. Huhn
N. Kicillof
E. Kirda
A. Kirshin
M. Klein
P. Knab
A.R. Kniep
M. Koch
B. Koenig
P. Kosiuczenko
S. Kremer-Davidson
J. Kuester
G. Lagorio
G. Lajios
L. Lamb
L. Lambers
R. Lanotte
M. Lohmann
A.P. Ludtke Ferreira
G. Lukácsy
M. Lund
I. Majzik
K. Mehner
S. Meier
H. Melgratti
B. Metzler
R. Meyer
M. Mikic-Rakic
S. Minocha
T. Modica
M. Moeller
M. Monga
D. Monteverde
Á. Moreira
A. Murphy
J. Oberleitner
J. Oldevik
A. Olivero
S. Olvovsky
J. Padberg
M. Pavlovic
G. Pintér
M.C. Pinto
M. Pinzger
A. Platzer
U. Prange
E. Quintarelli
R. Quites Reis
L. Rabelo
O.W. Rahlff
G. Rangel
L. Rapanotti
H. Rasch
A. Rausch
G. Reggio
G. Reif
R. Reussner
M. Ribaudo
J. Rubin
L. Samuelis
A. Schaefer
F. Schapachnik
Y. Shaham-Gafny
N. Sharygina
A. Solberg
J. Stafford
T. Staijen
S. Sutton
G. Toffetti Carughi
A. Toval
S. Ur
M. Urtasun
J. van der Linden
M. van der Meulen
G. van Dijk
D. Varró
G. Varró
J. Whitehead
S. Yovine
E. Zucca

Table of Contents

Invited Contributions

Distributed Systems

Orthogonal Process Activities

Behavioral Models and State Machines

Empirical Studies

Requirements and Design

Model-Based Development

Validation and Verification

Tool Demonstrations

Software Evolution

A Programming Model for Service Oriented Applications

Francisco Curbera

IBM Research, USA
curbera@us.ibm.com

Service oriented computing (SOC) and service oriented architectures introduce a model for distributed software components. Full inter-component interoperability, based on Web services standards, is a core assumption of the SOC model. SOC, as a platform independent approach to software development and management, is not limited to a particular distributed computing stack (Web services), since the benefits of a distributed component model extend to legacy protocols and platforms as well. Web services has successfully stressed the notion that implementation characteristics should be decoupled from interoperability concerns, and has focused on defining an XML based interoperability stack. SOC is directly concerned with the implementation and management of service oriented applications and stresses the ability to incorporate multiple runtimes and programming models into an architecture of distributed software components.

The Service Component Architecture (SCA) is the first realization of SOC as an explicit component model. Just as and Web Services provide the common abstraction of interoperability concerns, SCA provides a common abstraction of implementation concerns. SCA introduces a common notion of service components, service types and service implementations as well as an assembly model for service oriented applications. SCA's goal is to be able to accommodate multiple implementation platforms into a single set of component oriented abstractions. J2EE, BPEL4WS, COBOL, SQL or XML components are only part of the possible implementation artifacts that SCA intends to support. Portability of component assemblies and implementations is an important concern of SCA. SCA is already backed by a Java open source initiative in Apache.

An initiative so ambitious necessarily raises many open issues. Foremost among them is the formalization of an SCA runtime model sufficiently complete to ensure portability of implementations, but at the same time generic enough that it can be supported by multiple platforms and programming models. Once an SCA runtime model is defined, the question arises of whether a "native SCA" platform would be able to provide better support for the execution and deployment of SOC applications. Other significant issues include the possibility of formalizing the component and assembly models beyond their current state, and the support for non-functional requirements and capabilities in the definition and assembly of components.

This talk will review the motivation and major elements of the SCA model, and will discuss the main open issues surrounding the SCA effort.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, p. 1, 2006.

Software Engineering: Emerging Goals and Lasting Problems

Carlo Ghezzi

Dipartimento di Elettronica e Informazione - Politecnico di Milano,
Piazza L. da Vinci 32, I-20133 Milano, Italy
carlo.ghezzi@polimi.it

Software has been evolving from pre-defined, monolithic, centralized architectures to increasingly decentralized, distributed, dynamically composed federations of components. Software processes have been evolving along similar lines, from pre-specified sequential work- flows to decentralized and multi-organization endeavors. The organizations to which software solutions are targeted have also been evolving from highly structured corporates to agile and networked enterprises. All this is affecting the way software is engineered (i.e., conceived, architected, and produced). New difficult challenges arise, while old fundamental problems are still with us. The talk surveys this evolution and tries to identify achievements, challenges, and research directions.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, p. 2, 2006.

GPSL: A Programming Language for Service Implementation

Dominic Cooney, Marlon Dumas, and Paul Roe

Queensland University of Technology, Australia
{d.cooney, m.dumas, p.roe}@qut.edu.au

Abstract. At present, there is a dichotomy of approaches to supporting web service implementation: extending mainstream programming languages with libraries and metadata notations vs. designing new languages. While the former approach has proven suitable for interconnecting services on a simple point-to-point fashion, it turns to be unsuitable for coding concurrent, multi-party, and interrelated interactions requiring extensive XML manipulation. As a result, various web service programming languages have been proposed, most notably (WS-)BPEL. However, these languages still do not meet the needs of highly concurrent and dynamic interactions due to their bias towards statically-bounded concurrency. In this paper we introduce a new web service programming language with a set of features designed to address this gap. We describe the implementations in this language of non-trivial scenarios of service interaction and contrast them to the corresponding BPEL implementations. We also define a formal semantics for the language by translation to the join calculus. A compiler for the language has been implemented based on this semantics.

1 Introduction

There is an increasing acceptance of Service-Oriented Architectures as a paradigm for software application integration. In this paradigm, independently developed and operated applications are exposed as (web) services that are then interconnected using standard protocols and languages [1]. While the technology for developing basic services and interconnecting them on a point-to-point basis has attained some maturity, there remain open challenges when it comes to implementing service interactions that go beyond simple sequences of requests and responses or that involve many participants.

A number of recent and ongoing initiatives aim at tackling these challenges. These initiatives can be classified into conservative extensions to mainstream programming languages and novel service-oriented programming languages. The former provide metadata-based extensions for web service development on top of object-oriented programming languages. For example Microsoft Web Services Extensions, Windows Communication Foundation, Apache Axis and JSR-181, can be placed in this category. While these extensions are suitable for dealing with bilateral interactions and simple forms of concurrency and correlation,

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 3–17, 2006.

capturing complex interactions with these libraries remains daunting. On the other hand, a number of service-oriented languages have been proposed, ranging from research proposals (e.g. XL [2, 3]) down to standardisation initiatives, most notably the Business Process Execution Language for Web Services (BPEL) [4].

BPEL facilitates the development of services that engage in concurrent interactions and incorporates a declarative correlation mechanism, thus addressing some limitations of bespoke conservative language extensions. Nonetheless, it fails to provide direct support for typical service interaction scenarios. In [5], a number of patterns of service interaction are proposed. It is shown that while BPEL directly supports the most basic of these patterns, it fails to address the needs of more complex scenarios. In particular, BPEL has problems dealing with one-to-many interaction scenarios with partial synchronisation especially when the set of partner services is not known in advance.

The analysis of BPEL in [5] suggests that web service implementation requires novel programming abstractions for dealing with advanced forms of concurrency, synchronisation, and message correlation. Accordingly, this paper presents a programming language, Gardens Point Service Language (GPSL), that integrates concepts and constructs from join calculus [6], a declarative correlation mechanism with greater flexibility than BPEL's one, and direct support for complex XML data manipulation. Specifically, GPSL incorporates:

- Dedicated messaging constructs, both for interacting with the other services via SOAP, and for structuring the internal implementation of services
- A stratified integration of XQuery [7] expressions with imperative constructs.
- A join calculus-style approach to concurrent web service messaging, and an embodiment of this concurrency style as a programming language construct.
- An approach to message correlation that provides direct support for both point-to-point and one-to-many web service conversations [8].

A compiler implementation of GPSL can be found in [9]. The suitability of GPSL has been tested by implementing a number of scenarios, ranging from simple scenarios (e.g. an Amazon.com Queue Service client [10]) to scenarios corresponding to the more complicated service interaction patterns of [5]. In this paper, we sketch the implementations of three of these patterns.

The paper is structured as follows: Section 2 provides an overview of GPSL. Next, Section 3 describes the abstract syntax and formal semantics of GPSL. Section 4 illustrates how the language supports advanced service interaction patterns. Section 5 then briefly describes the compiler implementation of GPSL focusing on the code generation. Finally, Section 6 reviews related work while Section 7 concludes.

2 Overview of GPSL

To illustrate the basic features of GPSL, we consider the implementation of a simple 'echo' service and its client:

```
declare interface Echo {
  declare operation Shout in action = 'urn:echo:shout' out
}
declare service EchoService implements Echo {
  Shout($doc, Reply) { Reply($doc) }
}
declare service EchoClient {
  do {
    let $x := 'soap.tcp://localhost:4000/echo' in
    $x: Shout(element Say { 'Hello' }, Done)
  }
  Done($doc) { (: comment -- do nothing :) }
}
```

GPSL has explicit contract and service declaration elements. Metadata from contracts are used by the compiler to provide types to operations. For example, the *Echo* contract has one operation, *Shout. Shout* is declared as an *in-out* operation and by convention in GPSL has two parameters: one for data, and the other for a channel to send the reply on. When a *Shout* operation message is received, in the case of *EchoService,* or sent, in the case of the *EchoClient,* the first parameter is bound to the body of the SOAP envelope and the second parameter is bound to the WS-Addressing (WS-A) reply-to SOAP header.

EchoService declares *implements Echo* and includes a block guarded by a label *Shout* that takes two parameters. The *Shout* label refers to an operation in the *Echo* contract, so whenever the service receives a message with SOAP action *urn:echo:shout* the service executes the corresponding block of code. The language enforces a convention where variables bound to XML data are prefixed with a *$.* The *$doc* parameter is bound to the body of the SOAP message and the *Reply* parameter is bound to the WS-A reply-to header. *Reply,* although derived from XML in the SOAP envelope, describes the capability for sending a message and we do not prefix it with *$. Reply* is opaque and the capability can only be passed to another service or exercised to send a message. The syntax for sending a message is to write the channel variable and a parameter list in parentheses. In this example, *EchoService* sends in the reply the data it received in the request.

The data model of GPSL has two kinds of values: XML data, such as the element *Say,* and channels, such as *Reply.* All XML expressions in GPSL are XQuery expressions. For example, *element Say* is an example of the XQuery computed element constructor. This ability to construct new XML data distinguishes XQuery from the less powerful XPath. However XQuery alone is not sufficient for implementing services because it is a pure functional language with no messaging constructs. Moreover, there are some semantic tensions between XQuery's flexible evaluation semantics and messaging, because it is difficult to determine when a message will be sent or received. To avoid these tensions, GPSL is based on a stratified approach in which imperative constructs are used for messaging whereas XQuery is used for expressions.

Now let us consider the implementation of *EchoClient.* It contains a *do* block; *do* blocks are executed when a service starts up and can initialise state like a constructor in an object-oriented language. Here *EchoClient* sends a *Shout* message. For the first parameter it constructs an element *Say* that contains the text *Hello.* By convention, the second parameter becomes the WS-A reply-to header. Here *EchoClient* provides the label of a block, *Done,* as the second parameter. *Done* is a "private" label of *EchoClient,* and does not refer to any operation in a contract.

Messaging in GPSL is asynchronous; this encourages programmers to write services that make concurrent requests rather than sequences of request/responses, although this RPC programming style is also possible in GPSL. GPSL's means of spawning concurrent threads derives from asynchronous messaging. Using a private label to send an internal message starts the corresponding block of code which is executed concurrently with subsequent instructions.

```
  ...
  M(); (: sends local message, asynchronously :)
  ...  (: subsequent instructions go here :)
}

M() {
  (: this code executes concurrently when a message is sent on M :)
}
```

Synchronisation is achieved through blocks of code guarded by multiple labels. Such multi-label guards are called *concurrency patterns* and are inspired by the join calculus. A block of code guarded by a concurrency pattern is executed when messages are available on all labels. For example, in the following code snippet, local messages *ResultA* and *ResultB* are sent in two different blocks of code $A()$ and $B()$ which we assume are executed concurrently (although their spawning is not shown). When both messages are available, then the rule at the bottom is reduced and the corresponding block of code is executed.

```
A() {
  ...
  ResultA(...) (: produce message ResultA :)
}
B() {
  ...
  ResultB(...) (: produce message ResultB :)
}
ResultA($a) & ResultB($b) (* this is a join pattern *) {
  (: executed when ResultA and ResultB are available :)
}
```

3 Syntax and Semantics

The syntax of GPSL statements and expressions is shown in Figure 1. For space reasons we focus on statements and expressions omitting the *service* and *contract*

$S,\ T ::=$		statement
	ϵ	empty
	S **;** T	sequence
	if E **then** S **else** T **end**	conditional
	let v **:=** E **in** S	let-binding
	for v **in** E **do** S **end**	iteration
	$E(G, [rc])$	send (2nd argument may be used for reply channel)
	E **:** $m(G, \cdots)$	endpoint send
	def D **in** S	receive rules
$D,\ F ::=$		definitions
	J **{** S **}**	receive rule
	$D\ F$	composition
$J,\ K ::=$		pattern
	$x(y, \cdots)$	internal message receive
	receive y **where** E	external message receive
	$x(y)$[**where** E]	contract receive: x is an "in" operation defined in a contract
	$x(y, rc)$[**where** E]	contract receive: x is an "in-out" operation defined in a contract (rc stands for "reply channel")
	J **&** K	synchronisation
$E,\ G ::=$		expression
	m	label
	$\cdots$	XQuery expression

Where $m,\ v,\ x,\ y$ and rc are identifiers.

Fig. 1. Abstract syntax of the imperative GPSL statements

elements; these elements provide metadata about the SOAP action and message exchange patterns of operations and do not have a direct operational semantics.

We sketch the semantics of GPSL in Figure 2 via an operational encoding in the join calculus [6]. Since GPSL's concurrency feature is based directly on the join calculus this encoding is often straightforward syntax translation.

For our encoding we assume a join calculus with XQuery expressions and values. Where XQuery has flexible evaluation semantics related to laziness/strictness and raising errors, GPSL needs predictable behaviour for message sending. We introduce an explicit channel, *eval,* to specify precisely when XQuery evaluation occurs. *eval* forces XQuery evaulation in its first argument and passes the result on its second argument. *cond,* for implementing conditionals, is like *eval* except it chooses a continuation based on the result.

For sending messages on internal channels (rule "Int. Snd" in Figure 2) we only give the encoding of the single-argument case. Other arities follow the same pattern, where the message receiver and arguments are evaluated left-to-right. Likewise, for sending messages to other services (Ext. Snd,) we only give the case when the operation is expected to reply, where by convention in GPSL the first argument becomes the body of the message, and the second argument is

Empty:	$[\![\epsilon]\!]$	$\rightarrow 0$
Seq:	$[\![S;T]\!]$	$\rightarrow [\![S]\!].[\![T]\!]$
If:	$[\![\textbf{if } E \textbf{ then } S \textbf{ else } T \textbf{ end}]\!]$	$\rightarrow \textbf{def } t\langle\rangle \rhd [\![S]\!] \mid f\langle\rangle \rhd [\![T]\!] \textbf{ in } cond\langle E, t, f\rangle$
Let:	$[\![\textbf{let } v := E \textbf{ in } S]\!]$	$\rightarrow \textbf{def } s\langle v\rangle \rhd [\![S]\!] \textbf{ in } eval\langle E, s\rangle$
For:	$[\![\textbf{for } v \textbf{ in } E \textbf{ do } S \textbf{ end}]\!]$	$\rightarrow \textbf{def } test\langle es, s\rangle \rhd$ $\quad \textbf{def } t\langle\rangle \rhd$ $\quad\quad \textbf{def } hd\langle e\rangle \rhd$ $\quad\quad\quad \textbf{def } tl\langle es\rangle \rhd s\langle e, es\rangle \textbf{ in}$ $\quad\quad\quad eval\langle es[position() > 1], tl\rangle \textbf{ in}$ $\quad\quad eval\langle es[position() = 1], hd\rangle \textbf{ in}$ $\quad \textbf{def } f\langle\rangle \rhd 0 \textbf{ in}$ $\quad cond\langle es = nil(), t, f\rangle \textbf{ in}$ $\textbf{def } s\langle v, es\rangle \rhd [\![S]\!].test\langle es, s\rangle \textbf{ in}$ $\textbf{def } init\langle es\rangle \rhd test\langle es, s\rangle \textbf{ in}$ $eval\langle E, init\rangle$
Int. Snd:	$[\![E(G)]\!]$	$\rightarrow \textbf{def } receiver\langle e\rangle \rhd$ $\quad \textbf{def } actual_1\langle f\rangle \rhd e\langle f\rangle \textbf{ in}$ $\quad eval\langle G, actual_1\rangle \textbf{ in}$ $eval\langle E, receiver\rangle$
Ext. Snd:	$[\![E{:}m(G, H)]\!]$	$\rightarrow [\![\textbf{let } receiver := E \textbf{ in}$ $\textbf{let } actual := G \textbf{ in}$ $\textbf{let } reply := H \textbf{ in}$ $\textbf{let } id := gensym \textbf{ in}$ $\textbf{def receive } env$ $\quad \textbf{where } Header/RelatesTo = id \{$ $\quad reply(env)$ $\} \textbf{ in}$ $send(receiver, m_{action}, id, actual)]\!]$
Recv:	$[\![\textbf{def } D \textbf{ in } S]\!]$	$\rightarrow \textbf{def } [\![D]\!] \textbf{ in } [\![D]\!]_{Init}.[\![S]\!]$
Reaction:	$[\![J\{S\}]\!]$	$\rightarrow [\![J]\!] \rhd [\![S]\!]$
Composition:	$[\![D\ F]\!]$	$\rightarrow D \wedge F$
Synch:	$[\![D\ \&\ F]\!]$	$\rightarrow D \mid F$
Int. Recv:	$[\![x(y)]\!]$	$\rightarrow x\langle y\rangle$
Ext. Recv:	$[\![\textbf{receive } y \textbf{ where } E]\!]$	$\rightarrow x\langle y\rangle, x$ is fresh
Contract Recv:	$[\![x(y) \textbf{ where } E]\!]$	$\rightarrow x\langle y\rangle$
Init Reaction:	$[\![J\{S\}]\!]_{Init}$	$\rightarrow [\![J]\!]_{Init}$
Init Comp.:	$[\![D\ F]\!]_{Init}$	$\rightarrow [\![D]\!]_{Init}.[\![F]\!]_{Init}$
Init Synch:	$[\![D\ \&\ F]\!]_{Init}$	$\rightarrow [\![D]\!]_{Init}.[\![F]\!]_{Init}$
Init Int. Recv:	$[\![x(y)]\!]_{Init}$	$\rightarrow \mathbf{0}$
Init Ext Recv:	$[\![\textbf{receive } y \textbf{ where } E]\!]_{Init}$	$\rightarrow \textbf{def } x_{test}\langle y, t, f\rangle \rhd cond\langle E, t, f\rangle \textbf{ in}$ $subscribe\langle x, x_{test}\rangle$
Init Contract Recv:	$[\![x(y) \textbf{ where } E]\!]_{Init}$	$\rightarrow [\![\textbf{def receive } env$ $\quad \textbf{where } Header/Action = x_{action} \textbf{ and}$ $\quad\quad E \{\ x(env)\ \} \textbf{ in}$ $\cdots]\!]$, for the first occurence of $x(y)$ **where** E

Fig. 2. Partial semantics by translation into join calculus

a channel to use for replies. It is the metadata from a contract element that dictates whether a reply is expected and the SOAP action m_{action}. Correlating replies involves generating a new message ID, establishing a closure to listen for incoming messages with a matching message ID, and then sending the message. We write *send* for this latter step; *send* formats a SOAP envelope and sends it over the network.

Definitions and patterns follow predictable syntactic translation, except for external message receive which has no parallel in the join calculus. External message receive is responsible for marshalling SOAP messages received from the outside world into a GPSL program. This raises the important semantical issue of precisely what point in a program messages are delivered *to*. GPSL is more flexible and powerful than most contemporary programming languages in that it supports a *where* clause for filtering incoming messages. This feature is akin to filtering capabilities in message-oriented middleware and enables, among other things, to correlate any sent or received message with follow-up messages.

To encode external message receives, we create fresh internal channels and bind them to the SOAP messaging machinery via a message to *subscribe* when the closure is created. *subscribe* is a global internal channel with a complete join-calculus definition in Figure 3. This gives a precise semantics to receiving messages in GPSL: there is no race condition between receiving and sending messages in a closure as a closure is created, because of the continuation k threaded through *subscribe*, which is important for the correctness of closures initiating conversations; messages are routed into matching closures; concurrently active *receive* statements cause a runtime error if they compete for a particular message; and messages that have no active *receive* to process them are silently dropped.

There is syntactic sugar for receiving messages from an operation of an implemented contract (Init Ctrct Recv) which includes a test against the SOAP action specified in the contract. Our translation omits one detail in that the *receive* clause constructed for an *in-out* operation also creates a channel carrying the reply. The translation in Figure 2 is for an *in* operation.

$$
\begin{array}{l}
\textbf{def}\ subscribe\langle msg, predicate, k\rangle | subscribers\langle f\rangle \triangleright \\
\quad \textbf{def}\ g\langle x, found, done\rangle \triangleright \\
\qquad \textbf{def}\ true\langle\rangle \triangleright found\langle msg, f\rangle\ \textbf{in} \\
\qquad \textbf{def}\ false\langle\rangle \triangleright f\langle x, found, done\rangle\ \textbf{in} \\
\qquad predicate\langle x, true, false\rangle\ \textbf{in} \\
\quad subscribers\langle g\rangle.k\langle\rangle \\
\wedge \quad external\langle env\rangle | subscribers\langle f\rangle \triangleright \\
\qquad subscribers\langle f\rangle | \\
\qquad \textbf{def}\ done\langle\rangle | single\langle msg\rangle \triangleright msg\langle env\rangle\ \textbf{in} \\
\qquad \textbf{def}\ fail\langle msg, k\rangle \triangleright \text{error}\ \textbf{in} \\
\qquad \textbf{def}\ found\langle msg, k\rangle \triangleright single\langle msg\rangle.k\langle env, fail, done\rangle\ \textbf{in} \\
\qquad f\langle env, found, done\rangle\ \textbf{in} \\
\textbf{def}\ nil\langle x, found, done\rangle \triangleright done\langle\rangle\ \textbf{in} \\
subscribers\langle nil\rangle
\end{array}
$$

Fig. 3. Join-calculus definition of *subscribe*

4 Service Interaction Patterns in GPSL

In this section, we compare GPSL with BPEL by implementing scenarios corresponding to two of the service interaction patterns of [5]. Using the nomenclature and numbering of [5] we have chosen: one-to-many send/receive (pattern 7), and contingent requests (pattern 8). We choose not to illustrate patterns 1 to 4 since they correspond to simple point-to-point interactions and do not put forward significant differences between GPSL and other service-oriented programming languages such as BPEL; patterns 5 and 6 are partly subsumed by pattern 7; pattern 9 makes appeal to similar features as patterns 4 and 7; pattern 10 deals with transactional issues beyond the scope of GPSL and BPEL; and patterns 11 through 13 deal with interconnecting groups of services rather than implementing individual ones.

4.1 One-to-Many Send-Receive

We consider an interaction pattern where a service sends messages and collects responses before continuing. In this example we implement a broker service that solicits bids from a set of bidders, and collects responses, keeping track of the best (in this example, lowest) bid received. Bids are collected until a time-out occurs.

```
declare interface BrokerContract {
  declare operation InitiateAuction in action = 'urn:broker:init';
  ...
}

declare service Broker implements BrokerContract {
  InitiateAuction($env) {
    (: solicit bids :)
    for $bidder in $env/Bidders do
        $bidder: SolicitBid($env/Item, Reply)
    done;
    OutstandingBids(util:length($env/Bidders));

    (: start timer :)
    let $timeout := 'soap.inproc://timer' in
    $timeout: Time(10000, TimedOut);

    NoBids()
  }
  OutstandingBids($n) & Reply($bid) & NoBids() {
    Winning($bid);
    Decrement($n)
  }
  OutstandingBids($n) & Reply($bid) & Winning($best) {
    if xs:decimal($bid/Amount) < xs:decimal($best/Amount) then
      Winning($bid)
    else
```

```
      Winning($best)
    end;
    Decrement($n)
  }
  Decrement($n) {
    let $n := xs:int($n) - 1 in
    if xs:int($n) = 0 then
      BiddingFinished()
    else
      OutstandingBids($n)
    end
  }
  BiddingFinished() & Winning($bid) { (: process winning bid :) }
  TimedOut() & Winning($bid)        { (: fault or process winning bid :) }
  TimedOut() & NoBids()             { (: fault :) }
  ...
}
```

This program sends n *SolicitBid* messages. Although the *for* loop is sequential, the *SolicitBid* messages are sent in a non-blocking manner. The first bid received consumes the *NoBids* message and becomes the winning bid. Subsequent bids are compared to the winning bid. Messages *OutstandingBids* and *WinningBid* are used to capture state. They carry data for the number of outstanding bids and the best bid received. It is possible to check that this service correctly treats concurrent bids because contention for the *WinningBid* message acts as a mutual exclusion.

In previous work [8], we have sketched a more complicated variant of this scenario where the service stops after either receiving the first n-out-of-m responses or after the time-out, whichever occurs first.

Coding the above scenario in BPEL is complicated by several factors. First, given that the set of partners to which bid requests are sent is not known in advance, dynamic addressing is required. In GPSL, this is achieved by treating channels as first-class citizens. In BPEL, dynamic addressing is possible but requires manual assignment of endpoint references to *partner links*. Second, BPEL lacks high-level constructs for manipulating collections. Thus, capturing this scenario requires the use of *while* loops and additional book-keeping. Third, there is no direct support in BPEL for interrupting the execution of a block when a given event (e.g. a timeout) occurs. To achieve this, it is necessary to combine an event handler with a fault handler, such that the event handler raises a fault when the nominated event occurs and the fault handler catches this artificially created fault. This causes the immediately enclosing scope to be stopped. In GPSL, such interruption can be achieved simply by adding a join pattern that matches the event in question (in this case, the timeout). Finally, a further complication arises if explicit correlation using *correlation sets* is necessary. In this case, the first message needs to be treated differently from the following messages (at least in BPEL 1.1) since the first message initialises the correlation set. BPEL pseudo-code for this scenario is given below. The full version of this

pseudo-code is considerably longer than the corresponding GPSL solution. The interested reader will find the full BPEL implementation of a similar scenario in the code repository of the service interaction patterns site.[1] This implementation comprises around 150 lines of BPEL code excluding comments, partly due to the verbosity of the XML syntax, but also because of the more fundamental drawbacks of BPEL mentioned above.

```
set partner link to the address of the first bidder;
send first bid request and initialise correlation set;
while (more partners)
   set partner link to the address of next bidder;
   send next bid request;
begin scope
   onAlarm timeLimit : throw timeoutFault
   catch timeoutFault : set flag to indicate time-out
   while (not stopCondition)
      receive a bid;
      update winning bid
end scope
(* process time-out or winning bid *)
```

4.2 Contingent Requests

In this pattern, a service sends a message and if a response is not received within a given timeframe, a message is sent to a second service, and so on. If while waiting for a response from the second service, the first service happens to respond, this response is accepted and the response from the second service is no longer needed. This implements a fail-over process. An example of this pattern is a conference that provides redundant services to accept a paper submission. The client submits the paper via the first service, and if a response is not received within ten seconds, it submits the paper via the second service, and so on. Here is the implementation of the "client" in GPSL.

```
declare variable $timeout := 'soap.inproc://timer';
declare interface PaperSubmission {
  declare operation Submit in action = 'urn:paper:submit' out
}
declare service PaperSubmitter {
  do {
    let $submission-points := element Point { ... }, ... in
    let $paper := ... in
    Submit($paper, $submission-points)
  }
  Submit($paper, $submission-points) {
    let $uri := $submission-points[1] in
    let $submission-points := $submission-points[position()>1] in
```

[1] See code sample "One-to-many send/receive with dynamically determined partners" at http://www.serviceinteraction.com

```
    $uri: Submit($paper, Response);
    $timeout: Time(10000, TimedOut);
    Waiting($paper, $submission-points)
  }
  Waiting($paper, $submission-points) & Response($doc) { (: success! :) }
  Waiting($paper, $submission-points) & TimedOut($doc) {
    (: submit to the next server :)
    Submit($paper, $submission-points)
  }
}
```

The *PaperSubmitter* service has knowledge of a list of services that accept submissions. *Submit* strips a URI from the head of the list and sets up a race between *Response* and *TimedOut* messages. If a *TimedOut* message is available first, *PaperSubmitter* submits to the next service. After a response is received from any of the contacted services, *PaperSubmitter* does not wait for other responses, unless the code in the "success" block issues a new *Waiting* message.

This example shows the *PaperSubmitter* service interacting with a timer service at a known URI through *Time* (an operation to arm the timer) and *TimedOut* (an internal channel that receives time-out messages from the timer). This timer service does not need to be located outside *PaperSubmitter*'s memory space. Our implementation supports an efficient in-process transport for SOAP messaging, namely *soap.inproc*. This way, we can extend the language by adding services implemented in (e.g.) C#, rather than adding new constructs for every required feature. This is similar to a library, except that GSPL's library calling convention is based on messaging rather than function or method calls.

Capturing this example in BPEL in all its details is complicated by two factors: (i) the lack of direct support for interruptions due to an event as discussed previously; and (ii) the lack of support for maintaining an a priori unknown number of conversations in parallel. Indeed, this pattern puts forward a case where a requester may start a new conversation with a partner, but keep another ongoing conversation alive. There is only one construct in BPEL that supports an unbounded number of threads to be entertained concurrently: event handlers. However, using event handlers to capture the scenario at hand leads to an unintuitive solution. In this solution, the code for submitting a paper to a given server is embedded in an event handler. To start this event handler for the first server, the process sends a message to itself. This starts a first instance of the event handler. This event handler is terminated if a response is received (to do so, a fault indicating this is thrown). If a time-out occurs within this first instance of the event handler, the process sends a second message to itself to activate a second instance of the event handler (without stopping the previous instance since a late response from the first server may still arrive). This process of starting new instances of the event handler continues until a response is received or all servers have been tried.

The BPEL pseudo-code for this scenario is given below. Again, the full BPEL code is considerably more verbose, partly due to the need to define and configure the partner link through which the process sends messages to itself. The full

BPEL code for a similar scenario available at the serviceinteraction.com site comprises around 120 lines of code.

```
responseReceived := false
begin scope
  onMessage X :
    begin scope
      onAlarm timeLimit :
         if (more servers) send a message of type X to myself
         else throw allServersTimedOut
      catch allServersTimedOut: do nothing (* terminates scope *)
      catch responseReceived: do nothing (* terminates scope *)
      send request to next server and wait for response;
      responseReceived := true;
      throw responseReceived
    end scope
  send a message of type X to myself
end scope
if (not responseReceived) (* deal with case where no response received *)
```

5 Code Generation

We have prototyped a compiler that produces Microsoft Intermediate Language (MSIL), from GPSL programs. MSIL is similar to Java byte code although differing from it in various respects. Despite these differences though, our prototype proves that the feasibility of compiling GPSL for modern virtual machines.

Despite the novelty in the programming language, the compiler operates in traditional parsing, analysis, and code generation phases. The parser must handle XQuery for expressions. For our prototype we found ignoring XQuery direct constructors—the angle-brackets syntax for synthesizing XML which require special handling of whitespace—greatly simplifies parser development. Because syntactically simpler computed constructors can do the job of direct constructors, the expressive power of XQuery is unimpeded.

Most of the complexity in the compiler is in the code generator, and specifically in the creation of closures and in the delivery of messages sent on internal labels (i.e. messages from a service to itself). For each *def* we create a class with a method for each concurrency rule, a field for each captured variable, and a method and field for each label. This field holds a queue of pending messages; the method takes a message to that label, tests whether any rules are satisfied, and if so, calls the method for the rule. We perform the rule testing on the caller thread and only spawn a thread when a rule is satisfied, which avoids spawning many threads. The rule testing follows the join calculus semantics and the definition for the *subscribe* reaction rule given in Figures 2 and 3 for internal and external messages respectively.

We do not compile XQuery expressions because implementing an XQuery compiler is a daunting task. Instead we generate code to call an external XQuery

library at runtime. One critical criterion for the programming language implementer integrating an XQuery implementation is how that XQuery implementation accepts external variables and provides results. GPSL requires access to expression results as a sequence of XQuery data model values—which is distinctly different from an XML document—to behave consistently with XQuery when those values that are used later in subsequent expressions. We use an interoperability layer over the C API of Galax[2], which has exactly the kind of interface for providing external values and examining results that we want. Our biggest complaint about Galax is that evaluating expressions must be serialized because Galax is non-reentrant.

GPSL programs also rely on the Microsoft Web Services Extensions[3] (WSE) for SOAP messaging. WSE has a low-level messaging interface which is sufficient for GPSL's needs except for the fact that WSE does not support SOAP RPC/encoded. In the case of synchronous operations, the GPSL compiler generates some bookkeeping code to make SOAP over synchronous-HTTP work using the WSE messaging interface.

6 Related Work

Mainstream approaches to web service implementation are based on the use of Java, C#, and C++ in conjunction with libraries, such as Axis, and metadata annotations as in JSR181 and Windows Communication Foundation. Putting aside the mismatch between object-oriented and XML-based data manipulation, this approach has proven fairly suitable for programming point-to-point service interactions. However, it does not properly serve the requirements of multilateral interactions, especially those requiring partial synchronisation and message correlation beyond simple "request-response" scenarios. Message passing interfaces, like MPI [11], alleviate some of these issues, but even MPI's scatter and gather primitives assume barrier synchronisation and message correlation requires careful programming.

BPEL departs from mainstream web service implementation approaches by providing an XML data model, a set of message exchange primitives, concurrency constructs inspired from workflow languages, and a message correlation mechanism based on lexically scoped "static" variables. However, while BPEL supports high-level concurrency and barrier synchronisation constructs for fixed numbers of threads (for example through the "flow" construct), it does not support partial synchronisation nor unbounded numbers of threads, and thus, the expression of patterns such as one-to-many send-receive, multi-responses and contingent requests is cumbersome. Also, support for message correlation in BPEL is limited: BPEL's correlation sets can not be used to capture the type of correlation required by the one-to-many send-receive pattern.

Similar comments apply to XL, which provides a correlation mechanism suitable for 1:1 conversations, but not for $1 : n$ scenarios. Similarly, XL is suitable for

[2] http://www.galaxquery.org
[3] http://msdn.microsoft.com/webservices/building/wse

barrier synchronisation of conversations but not for partial synchronisation. Finally, XL relies on in-place updates of XML nodes through extensions to XQuery, while GPSL adopts a stratified approach where XQuery is only used as an expression language, orthogonal to the imperative part of the language. A more detailed comparison of XL and an earlier version of GPSL can be found in [8].

GPSL draws one of its main constructs, concurrency patterns, from the join calculus. The join calculus has inspired several extensions of object-oriented programming languages with concurrency features, namely Join Java [12] and Polyphonic C# [13]. Compared to these languages, GPSL adds XML data manipulation, messaging and message correlation. An extension to Polyphonic C#, Cω[4], adds XML data manipulation, but retains the legacy object data model and does not have explicit support for messaging or message correlation.

7 Conclusion

We have presented the syntax and semantics of the GPSL language and illustrated its suitability for service implementation using scenarios corresponding to patterns identified elsewhere. This exercise showed how simple features based on SOAP messaging, join-calculus style declarative concurrency, and XQuery can be combined to implement non-trivial patterns of service interaction in a way that arguably leads to simpler solutions than in BPEL. Also, GPSL's formal semantics is much simpler than corresponding semantics of BPEL[5] thus providing a solid basis for program analysis. The compiler implementation of GPSL, especially with respect to compiling rules with *where* statements, mirrors the formal semantics.

GPSL integrates messaging, concurrency, and XML data manipulation cohesively. Examples of the cohesive fit are the interplay between sending messages and spawning concurrent threads on the one hand, and receiving messages and synchronising threads on the other; dynamic XML data describing message recipients; concurrency patterns describing thread-safe access to XML data; and the consistent treatment of inter- and intra-service messages. Sometimes the cohesion is imperfect. For example, channels and channel variables can not appear in arbitrary XQuery expressions. This is a deliberate restriction which provides a simple way to preserve strong typing for internal message sending, and to control when an internal channel has to be connected to the machinery for receiving SOAP messages from the outside world. However channels *are* reified to XML when they appear in a WS-A "reply-to" header.

GPSL could be extended to address other difficult aspects of service implementation such as transactions and faults. We expect to address these areas by leveraging the messaging and concurrency features, for example, by surfacing faults as messages. We also plan to introduce a garbage collection technique to reclaim resources when it is detected that a given message will not be consumed.

[4] http://research.microsoft.com/Comega
[5] For a semantics of BPEL see e.g. [14].

Acknowledgments. The first author completed this work while working at Microsoft. The second author is funded by a fellowship co-sponsored by Queensland Government and SAP.

References

1. Weerawarana, S., Curbera, F., Leymann, F., Story, T., Ferguson, D.: Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable-Messaging, and More. Prentice Hall (2005)
2. Florescu, D., Grünhagen, A., Kossmann, D.: XL: an XML programming language for Web service specification and composition. In: International World Wide Web Conference, Honolulu, HI, USA (2002)
3. Florescu, D., Grünhagen, A., Kossmann, D.: XL: A platform for Web services. In: Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, USA (2003)
4. Andrews, T., Curbera, P., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business process execution language for Web services version 1.1 (2003)
5. Barros, A., Dumas, M., Hofstede, A.: Service interaction patterns. In: Proceedings of the 3rd International Conference on Business Process Management, Nancy, France, Springer Verlag (2005) Extended version available at: http://www.serviceinteraction.com.
6. Fournet, C., Gonthier, G.: The reflexive chemical abstract machine and the join calculus. In: The 23rd ACM Symposium on Principles of Programming Languages (POPL). (1996) 372–385
7. Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., Siméon, J.: XQuery 1.0: An XML query language. W3C Working Draft (2005)
8. Cooney, D., Dumas, M., Roe, P.: A programming language for web service development. In Estivill-Castro, V., ed.: Proceedings of the 28th Australasian Computer Science Conference, Newcastle, Australia, Australian Computer Society (2005)
9. Cooney, D.: GPSL home page. WWW (2005) http://www.serviceorientation.com.
10. Cooney, D., Dumas, M., Roe, P.: Programming and compiling web services in GPSL. In: Proceedings of the 3rd International Conference on Service-Oriented Programming (ICSOC) – Short papers track, Amsterdam, The Netherlands, Springer (to appear) (2005)
11. Snir, M., Gropp, W.: MPI: The Complete Reference. 2nd edn. MIT Press (1998)
12. Itzstein, G.S., Kearney, D.: Applications of Join Java. In Lai, F., Morris, J., eds.: Seventh Asia-Pacific Computer Systems Architectures Conference (ACSAC2002), Melbourne, Australia, ACS (2002)
13. Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for C#. In Magnusson, B., ed.: Proceedings of the 16th European Conference on Object Oriented Programming (ECOOP), Malaga, Spain, 10–14 June 2002. Volume 2374 of Lecture Notes in Computer Science., Springer (2002) 415–440
14. Ouyang, C., Aalst, W., Breutel, S., Dumas, M., Hofstede, A., Verbeek, H.: Formal Semantics and Analysis of Control Flow in WS-BPEL (Revised Version). BPM Center Report BPM-05-13, www.bpmcenter.org (2005)

A Formal Approach to Event-Based Architectures

José Luiz Fiadeiro[1] and Antónia Lopes[2]

[1] Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, UK
jose@fiadeiro.org
[2] Department of Informatics, Faculty of Sciences, University of Lisbon,
Campo Grande, Lisboa 1749-016, Portugal
mal@di.fc.ul.pt

Abstract. We develop a formal approach to event-based architectures that serves two main purposes: to characterise the modularisation properties that result from the algebraic structures induced on systems by this discipline of coordination; and to further validate and extend the CommUnity approach to architectural modelling with "implicit invocation", or "publish/subscribe" interactions. This is a first step towards a formal integration of architectural styles.

1 Introduction

Event-based interactions are now established as a major paradigm for large-scale distributed applications (e.g. [1,3,5,10,12]). In this paradigm, components may declare their interest in being notified when certain events are published by other components of the system. Typically, components publish events in order to inform their environment that something has occurred that is relevant for the behaviour of the entire system. Events can be generated either in the internal state of the components or in the state of other components with which they interact.

Although Sullivan and Notkin's seminal paper [14] focuses on tool integration and software evolution, the paradigm is much more general: components can be all sorts of runtime entities. What is important is that components do not know the identity, or even the existence, of the publishers of the events they subscribe, or the subscribers of the events that they publish. In particular, event notification and propagation are performed asynchronously, i.e. the publisher cannot be prevented from generating an event by the fact that given subscribers are not ready to react to the notification.

Event-based interaction has also been recognised as an "abstract" architectural style, i.e. as a means of coordinating the behaviour of components during high-level design. The advantages of adopting such a style so early in the development process stem from exactly the same properties recognised for middleware: loose coupling allows better control on the structural and behavioural complexity of the application domain; domain components can be modelled independently and easily integrated or removed without disturbing the whole system.

However, in spite of these advantages and its wide acceptance, implicit invocation remains relatively poorly understood. In particular, its structural properties as an architectural style remain to be clearly stated and formally verified. One has to

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 18 – 32, 2006.

acknowledge the merit of several efforts towards providing methodological principles and formal semantics (e.g. [14]), including recent incursions on using model-checking techniques for reasoning about such systems [2,9]. However, we are still far from an accepted "canonical" semantic model over which all these efforts can be brought together to provide effective support and formulate methodological principles that can steer development independently of specific choices of middleware.

This paper makes another contribution in this direction by investigating how event-based interactions can be formalised in a categorical setting similar to the one that we developed in [7] for i/o-communication and action synchronisation (rendez-vous) around the language CommUnity. Our formalisation addresses the architectural properties, i.e. the discipline of decomposition and interconnection, not the notification and subscription mechanisms that support them. More precisely, it serves two main purposes. On the one hand, to characterise the modularisation properties that result from the algebraic structures induced on systems by this discipline of coordination. In particular, we justify a claim made in [14] about the externalisation of mediators: "Applying this approach yields a system composed of a set of independent and visible [tool] components plus a set of separate, or *externalised*, integration components, which we call *mediators*". Our interest is in investigating and assigning a formal meaning to notions such as "independent", "separate" and "externalised", and in characterising the way they can be derived from implicit invocation. On the other hand, we wish to further validate and refine the categorical approach that we have been developing to support architectural modelling by investigating how the "implicit invocation" architectural style can be captured as a coordinated category [6]. This is a first step towards a formal approach to the integration of architectural styles.

In section 2, we introduce our primitives for modelling publish/subscribe interactions using a minimal language in the style of CommUnity [7]. In section 3, we define the category over which we formalise our approach. We show how the notion of morphism can be used to identify components within systems and the way they can subscribe events published by other components. In section 4, we show how event bindings can be externalised and made explicit in configuration diagrams. In section 5, we give a necessarily brief account of how we can use the categorical formalisation to bring several architectural styles together.

2 Event-Based Designs

We model components that keep a local state and subscribe to a number of events. Upon notification that one such event has taken place, a component invokes one or more services. If, when invoked, a service is enabled, it is executed, which may change the local state of the component and publish new events.

We start discussing our approach by showing how we can model what is considered to be the "canonical" example of event-based interactions: the set-counter. We start with the design of a component *Set* that keeps a set *elems* of natural numbers as part of its local state. This component subscribes two kinds of events – *doInsert* and *doDelete* – each of which carries a natural number as a parameter. Two other kinds of

events – *inserted* and *deleted* – are published by *Set*. Each of these events also carries a natural number as a parameter.

As a component, *Set* can perform two kinds of services – *insert* and *delete*. These services are invoked upon notification of events *doInsert* and *doDelete*, respectively. When invoked, *insert* checks if the parameter of *doInsert* is already in *elems*; if not, it adds it to *elems* and publishes an *inserted* event with the same parameter. The invocation of *delete* has a similar behaviour.

```
design Set is
publish inserted
   par which:nat
publish deleted
   par which:nat
subscribe doInsert
   par which:nat
   invokes insert
   handledBy insert? ∧
      which=insert.lm
subscribe doDelete
   par which:nat
   invokes delete
   handledBy delete? ∧
      which=delete.lm
```

```
store elems: set(nat)
provide insert
   par lm:nat
   assignsTo elems
   guardedBy lm∉elms
   publishes inserted
   effects  elems'={lm}∪elems ∧
     inserted! ∧ inserted.which=lm
provide delete
   par lm:nat
   assignsTo elems
   guardedBy lm∈elms
   publishes deleted
   effects  elems'=elems\{lm} ∧
      deleted! ∧ deleted.which=lm
```

Even if the notation is self-explanatory, we need to discuss some of its features:

- When declaring the events that a component *subscribes*, we identify under *invokes* the services that may be invoked when a notification is received. Under *handledBy*, we specify the different ways in which a notification is handled, using *s?* to denote the invocation of service *s*.
- Parameter passing is made explicit through expressions within specifications. For instance, the clause *inserted.which=lm* in the definition of the effects of *insert* means that the event *inserted* is published with its parameter *which* equal to the value of the parameter *lm* of *insert*.
- Under *store* we identify the state variables of the component; state is local in the sense that the services of a component cannot change the state variables of other components.
- Through *assignsTo* we identify the state variables that a service may change and, through *publishes*, we identify the events that a service may publish.
- When specifying the *effects* of a service, *v'* denotes the value that state variable *v* takes after it is executed, and *e!* denotes the publication of event *e*.
- Through *guardedBy* we identify the enabling condition of a service, i.e. the set of states in which its invocation is accepted and the service is executed.
- Designs can be underspecifed, leaving room for further design decisions to be made during development. Therefore, we allow for arbitrary expressions to be used when specifying how parameters are passed, events are handled and services change the state.

Consider now the design of a system in which a counter subscribes *inserted* and *deleted* to count the number of elements in the set:

```
design Set&Counter is
store elems: set(nat),
  value:nat
publish&subscribe inserted
   par which:nat
   invokes inc
   handledBy inc?
publish&subscribe deleted
   par which:nat
   invokes dec
   handledBy dec?
subscribe doInsert
   par which:nat
   invokes insert
   handledBy insert? ∧
     which=insert.lm
subscribe doDelete
   par which:nat
   invokes delete
   handledBy delete? ∧
     which=delete.lm
provide insert
   par lm:nat
   assignsTo elems
   guardedBy lm∉elms
   publishes inserted
   effects  elems'={lm}∪elems ∧
     inserted! ∧ inserted.which=lm
provide delete
   par lm:nat
   assignsTo elems
   guardedBy lm∈elms
   publishes deleted
   effects  elems'=elems\{lm} ∧
     deleted! ∧ deleted.which=lm
provide inc
   assignsTo value
   effects  value'=value+1
provide dec
   assignsTo value
   effects  value'=value-1
```

We can keep extending the design by bringing in new components that subscribe given events. For instance, we may wish to keep a record of the sum of all elements of the set by adding an adder that also subscribes *inserted* and *deleted*.

```
design Set&Counter&Adder is
store elems: set(nat),
     value:nat, sum:nat
publish&subscribe inserted
   par which:nat
   invokes inc, add
   handledBy inc?
   handledBy add? ∧
     which=add.lm
publish&subscribe deleted
   par which:nat
   invokes dec,sub
   handledBy dec?
   handledBy sub? ∧
     which=sub.lm
subscribe doInsert
   par which:nat
   invokes insert
   handledBy insert? ∧
      which=insert.lm
subscribe doDelete
   par which:nat
   invokes delete
   handledBy delete? ∧
     which=delete.lm
provide insert
   par lm:nat
   assignsTo elems
   guardedBy lm∉elms
   publishes inserted
   effects  elems'={lm}∪elems ∧
     inserted! ∧ inserted.which=lm
provide delete
   par lm:nat
   assignsTo elems
   guardedBy lm∈elms
   publishes deleted
   effects  elems'=elems\{lm} ∧
      deleted! ∧ deleted.which=lm
provide inc
   assignsTo value
   effects  value'=value+1
provide add
   par lm:nat
   assignsTo sum
   effects  sum'=sum+lm
provide sub
   par lm:nat
   assignsTo sum
   effects  sum'=sum-lm
provide dec
   assignsTo value
   effects  value'=value-1
```

This example illustrates how we can declare more than one handler for a given event subscription. For instance, the event *inserted* has two handlers: one invokes *add*

and the other invokes *inc*. Both invocations are independent in the sense that they can take place at different times. This is different from declaring just one handler of the form $inc? \wedge add? \wedge which=add.lm$; such a handler would require synchronous invocation of both services. The latter is useful when one wants to make sure that separate state components are updated simultaneously, say to ensure that the values of *sum* and *count* apply to the same set of elements.

As a design of a system, *Set&Counter&Add* seems to be highly unstructured: we seem to have lost the original *Set*; and where is the *Counter*? and the *Adder*? In the next section, we show how *Set&Counter&Add* can be designed by interconnecting separate and independent components, including mediators in the sense of [14].

3 Structuring Event-Based Designs

In order to discuss the structuring of event-based designs, we adopt the categorical approach that we have been developing for architectural modelling [6,7]. In Category Theory, the structure of objects such as the designs introduced in the previous section is formalised in terms of *morphisms*. A morphism is simply a mechanism for recognising a component within a larger system.

In the examples discussed in the previous section, we used a number of data types and data type constructors. In order to remain independent of any specific language for the definition of the data component of designs, we assume a data signature $\Sigma=<D,\Omega>$, where D is a set (of sorts) and Ω is a $D^* \times D$-indexed family of sets (of operations), to be given together with a collection Φ of first-order sentences specifying the functionality of the operations. We refer to this data type specification by Θ.

From a mathematical point of view, designs are structures defined over signatures.

Definition: *A signature is a tuple* $Q=<V,E,S,P,T,A,B,G,H>$ *where*

- V *is a* D*-indexed family of finite sets (of state variables).*
- E *is a finite set (of events).*
- S *is a finite set (of services).*
- P *assigns to every service* $s \in S$ *and event* $e \in E$*, a* D*-indexed family of mutually disjoint finite sets (of parameters).*
- $T: E \rightarrow \{pub,sub,pubsub\}$ *is a function classifying events as published, subscribed, or both published and subscribed. We denote by* $Pub(E)$ *the set of events* $\{e \in E: T(e) \neq sub\}$ *and by* $Sub(E)$ *the set of events* $\{e \in E: T(e) \neq pub\}$*.*
- $A: S \rightarrow 2^V$ *is a function returning the write-frame (or domain) of each service.*
- $B: S \rightarrow 2^{Pub(E)}$ *is a function returning the events published by each service.*
- $G: Sub(E) \rightarrow 2^S$ *is a function returning the services invoked by each event.*
- H *assigns to every subscribed event* $e \in Sub(E)$*, a set (of handlers).*

The mapping P defines, for every event and service, the name and the type of its parameters. Every variable and parameter v is typed with a sort $sort(v) \in D$. The sets $V_{d \in D}$, E, S, $P_{s \in S}$ and $P_{e \in E}$ are assumed to be mutually disjoint. This is why the "official" name of, for instance, parameter *which* of event *inserted* is *inserted.which*.

We use T to classify events as *pub* (published only), *sub* (subscribed only) or *pub-sub* (both published and subscribed). For instance, in *Set&Counter&Adder (SCA):*

- E_{SCA}={inserted, deleted, doInsert, doDelete}
- T_{SCA}(inserted)=T_{SCA}(deleted)=pubsub; T_{SCA}(doInsert)=T_{SCA}(doDeleted)=sub
- $Sub_{SCA}(E)$={inserted, deleted, doInsert, doDelete}
- $Pub_{SCA}(E)$= {inserted, deleted}

And in *Set (S)* we have

- E_S={inserted, deleted, doInsert, doDelete}
- T_S(inserted)=T_S(deleted)=pub; T_S(doInsert)=T_S(doDeleted)=sub
- $Sub_S(E)$={doInsert, doDelete}
- $Pub_S(E)$= {inserted, deleted}

Events are published by services. We declare the events that each service may publish through the mapping B. For instance,

- B_S(insert)=B_{SC}(insert)=B_{SCA}(insert)={inserted}
- B_S(delete)=B_{SC}(delete)=B_{SCA}(delete)={deleted}

For every service s, another set $A(s)$ is defined that consists of the state variables that can be affected by instances of s. These are the variables indicated under *assignsTo*. For instance, A_S(insert)={elems}. We extend the notation to state variables so that $A(v)$ is taken to denote the set of services that have v in their write-frame. Hence, A_S(elems)={insert,delete}.

When a notification that a subscribed event has been published is received, a component reacts by invoking services. For every subscribed event e, we denote by $G(e)$ the set of services that may be invoked. For instance,

- G_S(doInsert)=G_{SC}(insert)=G_{SCA}(insert)={insert}
- G_{SC}(inserted)={inc}
- G_{SCA}(inserted)={inc,add}

Notice that the functions A, B, and G just declare the state variables, events and services that can be changed, published, and invoked, respectively. Nothing in a signature states how state variables are changed, or how and in which circumstances events are published or services invoked. In brief, signatures need to include all and only the typing information required for establishing interconnections. Hence, for instance, it is important to include in the signature information about which state variables are in the domain of which services but not the way services affect the state variables; it is equally important to know the structure of handlers for each subscribed event but not the way each subscription is handled. This additional information that pertains to the individual behaviour of components is defined in the *bodies* of designs:

Definition: *A design is a pair $<Q,\Delta>$ where Q is a signature and Δ, the body of the design, is a tuple $<\eta,\rho,\chi,>$ where:*

- *η assigns to every handler $h \in H(e)$ of a subscribed event $e \in Sub(E)$, a proposition in the language of V (state variables), the parameters of e, the services declared in G(e) and their parameters.*

- *ρ assigns to every service $s \in S$, a proposition in the language of V, the parameters of s, the primed variables in the domain of s, as well as the events – $B(e)$ – that may be published by the service and their parameters.*
- *γ assigns to every service $s \in S$, a proposition in the language of V (state variables) and the parameters of s.*

By "the language of X" we mean the first-order language generated by using X as atomic terms. Given this, the body of a design is defined in terms of:

- for every subscribed event *e*, a set – $H(e)$ – of *handling requirements* expressed through propositions $\eta(h)$ for every handler $h \in H(e)$. For instance, in *Set&Counter&Adder*, we have $H_{SCA}(inserted)$ given by two handlers whose requirements are *inc?* and $(add? \wedge inserted.which=add.lm)$. Every handling requirement (handling for short) is enforced when the event is published. Each handling consists of service invocations and other properties that need to be observed on invocation (e.g. for parameter passing) or as a precondition for invocation (e.g. in the case of filters for discarding notifications). A typical handling is of the form $\psi \supset (s? \wedge \phi)$ establishing that *s* is invoked with property ϕ if condition ψ holds on notification.
- for every service *s*, *an enabling condition* – $\gamma(s)$ – defining the states in which the invocation of *s* can be accepted. This is the condition that we specify under *guardedBy*.
- for every service *s*, a proposition – $\rho(s)$ – defining the *state changes* that can be observed due to the execution of *s*. As shown in the examples, this proposition may include the publication of events and parameter passing. This is the condition that we specify under *effects*.

The language over which propositions used in η, γ and ρ are written extends that used for the data type specification with state variables (and their primed versions in the case of ρ) as nullary operators. Qualified parameters of events and services are also taken as nullary operators. In the case of $\rho(s)$ this extension also comprises the events of $B(s)$ as nullary operators that represent the publication of the corresponding event. This is why $\rho_{SCA}(insert)$ includes the expression *inserted!* indicating the publication of the event *inserted*. In the case of $\eta(e)$ the extension includes services $a \in G(e)$ as nullary operators that represent their invocation, what we denote with *a?*.

As already mentioned, the structure of designs is captured through *morphisms*. These are maps between designs that identify ways in which the source is a component of the target. We define first how morphisms act on signatures:

Definition/Proposition: *A morphism $\sigma: Q_1 \to Q_2$ for $Q_1=<V_1,E_1,S_1,P_1,T_1,A_1,B_1,G_1,H_1>$ and $Q_2=<V_2,E_2,S_2,P_2,T_2,A_2,B_2,G_2,H_2>$ is a tuple $<\sigma_{st}, \sigma_{ev}, \sigma_{sv}, \sigma_{par\text{-}ev}, \sigma_{par\text{-}sv}, \sigma_{hr\text{-}ev}>$ where*

- *σ_{st}: $V_1 \to V_2$ is a function on state variables that preserves their sorts, i.e. $sort_2(\sigma_{st}(v))=sort_1(v)$ for every $v \in V_1$*
- *σ_{ev}: $E_1 \to E_2$ is a function on events that preserves kinds, i.e. $\sigma_{ev}(e) \in Pub(E_2)$ for every $e \in Pub(E_1)$ and $\sigma_{ev}(e) \in Sub(E_2)$ for every $e \in Sub(E_1)$, as well as invoked services, i.e. $\sigma_{sv}(G_1(e)) \subseteq G_2(\sigma_{ev}(e))$ for every $e \in Sub(E_1)$.*

- σ_{sv}: $S_1 \to S_2$ *is a function that preserves domains, i.e* $A_2(\sigma_{st}(v))=\sigma_{sv}(A_1(v))$ *for every* $v \in V_1$, *as well as published events, i.e.* $\sigma_{ev}(B_1(s)) \subseteq B_2(\sigma_{sv}(s))$
- $\sigma_{par\text{-}ev}$ *maps every event e to a function* $\sigma_{par\text{-}ev,e}$: $P_1(e) \to P_2(\sigma_{ev}(e))$ *that preserves the sorts of parameters, i.e.* $sort_2(\sigma_{par\text{-}ev,e}(p))=sort_1(p)$ *for* $p \in P_1(e)$
- $\sigma_{par\text{-}sv}$ *operates like* $\sigma_{par\text{-}ev}$ *but on service parameters*
- $\sigma_{hr\text{-}ev}$ *maps every subscribed event e to a function* $\sigma_{hr\text{-}ev,e}$: $H_1(e) \to H_2(\sigma_{ev}(e))$.

Signatures and their morphisms constitute a category ***SIGN.***

A morphism σ from Q_1 to Q_2 is intended to support the identification of a way in which a component with signature Q_1 is embedded in a larger system with signature Q_2. Morphisms map state variables, services and events of the component to corresponding state variables, services and events of the system, preserving data sorts and kinds. An example is the inclusion of *Set* in *Set&Counter&Adder*.

Notice that is possible that an event that the component subscribes is bound to an event published by some other component in the system, thus becoming *pubsub* in the system. This is why we have $T_S(inserted)=sub$ but $T_{SCA}(inserted)=pubsub$.

The constraints on domains imply that new services of the system cannot assign to variables of the component. This is what makes state variables "private" to components. As a result, we cannot identify components of a system by grouping state variables, services and events in an arbitrary way. For instance, we can identify a counter as a component of *Set&Counter&Adder* as follows. Consider the following design:

```
design Counter is                store value: nat
subscribe doInc                  provide inc
   invokes inc                      assignsTo value
   handledBy inc?                   effects  value'=value+1
subscribe doDec                  provide dec
   invokes dec                      assignsTo value
   handledBy dec?                   effects  value'=value-1
```

It we map *doInc* to *inserted* and *doDec* to *deleted*, we do define a morphism between the signatures of *Counter* and *Set&Counter&Adder*. Indeed, sorts of state variables are preserved, and so are the kinds of the events. The domain of the state variable *value* is also preserved because the other services available in *Set&Counter&Adder* do not assign to it.

Components are meant to be "reusable" in the sense that they are designed without a specific system or class of systems in mind. In particular, the components that are responsible for publishing events, as well as those that will subscribe published events, are not fixed at design time. This is why, in our language, all names are local and morphisms have to account for any renamings that are necessary to establish the *bindings* that may be required. For instance, the morphism that identifies *Counter* as a component of *Set&Counter&Adder* is not just an injection. Do notice that the binding also implies that *inserted* and *deleted* are subscribed within *Set&Counter&Adder*. As a result, our components are independent in the sense of [14]: they do not explicitly invoke any component other than themselves.

In order to identify components in systems, the bodies of their designs also have to be taken into account, i.e. the "semantics" of the components have to be preserved. We recall that we denote by Φ the specification of the data sorts and operations.

Definition/Proposition: *A superposition morphism* $\sigma:\langle Q_1,\Delta_1\rangle \to \langle Q_2,\Delta_2\rangle$ *consists of a signature morphism* $\sigma:Q_1 \to Q_2$ *such that:*

1. *Handling requirements are preserved: for every event* $e \in E_1$ *and handling* $h \in H_1(e)$, $\Phi \vdash \eta_2(\sigma_{hr\text{-}ev,e}(h)) \supset \underline{\sigma}(\eta_1(h))$
2. *Effects are preserved:* $\Phi \vdash (\rho_2(\sigma_{sv}(s)) \supset \underline{\sigma}(\rho_1(s))$ *for every* $s \in S_1$
3. *Guards are preserved:* $\Phi \vdash (\gamma_2(\sigma_{sv}(s)) \supset \underline{\sigma}(\gamma_1(s))$ *for every* $s \in S_1$

Designs constitute a category ***DSGN****. We denote by* ***sign*** *the forgetful functor from* ***DSGN*** *to* ***SIGN*** *that forgets everything from designs except their signatures.*

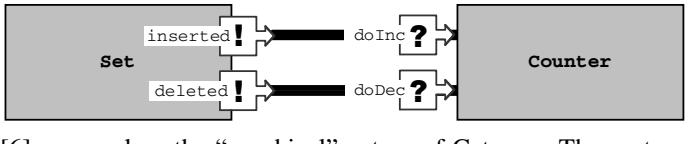
Notice that the first condition allows for more handling requirements to be added and, for each handling, subscription conditions to be strengthened. In other words, as a result of being embedded in a bigger system, a component that publishes a given event may acquire more handling requirements but also more constraints on how to handle previous requirements, for instance on how to pass new parameters.

It is easy to see that these conditions are satisfied by the signature morphisms that identify *Set* and *Counter* as components of *Set&Counter&Adder*. However, in general, it may not be trivial to prove that a signature morphism extends to a morphism between designs. After all, such a proof corresponds to recognising a component within a system, which is likely to be a highly complex task unless we have further information on how the system was put together. This is why it is important to support an architectural approach to design through which systems are put together by interconnecting independent components. This is the topic of the next section.

4 Externalising the Bindings

As explained in [7], one of the advantages of the categorical formalisation is that it supports a design approach based on superposing separate components (connectors) over independent units. These separate components are called mediators in [14]: for instance, *Set* as used for connecting a *Counter* and independent components that publish insertions and deletions. Morphisms, as defined in the previous section, enable the definition of such a design approach by supporting the externalisation of bindings.

For instance, using a graphical notation for the interfaces of components – the events they publish and subscribe, and the services that they can perform – we are able to start from separate *Set* and *Counter* components and superpose, externally, the bindings through which *Counter* subscribes the events published by *Set:*



Like in [6], we explore the "graphical" nature of Category Theory to model interconnections as "boxes and lines". In our case, the lines need to be accounted for by special components that perform the bindings between the event published by one component and subscribed by the other:

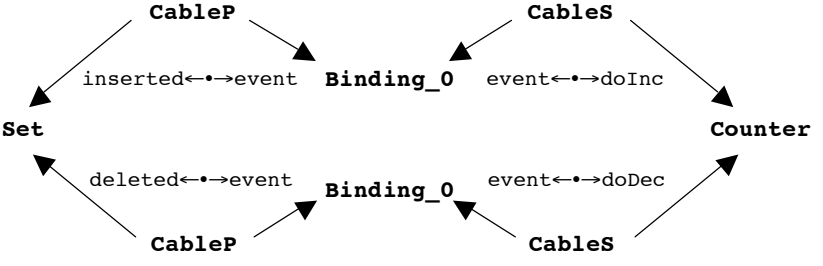
```
design Binding_0 is
publish&subscribe event
```

The binding has a single event that is both published and subscribed. The interconnection between *Set*, *Binding_0* and *Counter* is performed by an even simpler kind of component: cables that attach the bindings to the events of the components.

```
design CableP is
publish •
```

```
design CableS is
subscribe •
```

Because names are local, the identities of events in cables are not relevant: they are just placeholders for the projections to define the relevant bindings. Hence, we represented them through the symbol •. The configuration given above corresponds to the following diagram (labelled graph) in the category ***DSGN*** of designs:



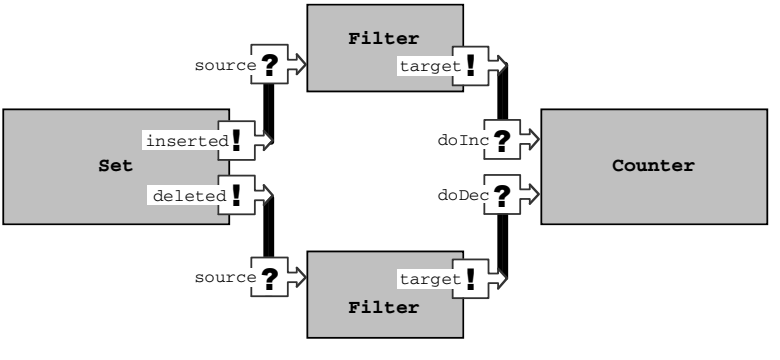
In Category Theory, diagrams are mathematical objects and, as such, can be manipulated in a formal way. One of the constructs that are available on certain diagrams internalises the connections in a single (composite) component. In the case above, this consists in computing the colimit of the diagram [6], which returns the design *Set&Counter* discussed in section 2. In fact, the colimit returns the morphisms that identify both *Set* and *Counter* as components of *Set&Counter*.

Bindings can be more complex. Just for illustration, consider the case in which we want to count only the even elements that are inserted. Instead of using *Binding_0* we would use a more elaborate connector *Filter* defined as follows:

```
design Filter is
publish&subscribe target
provide service
   publishes target
   effects target!
```

```
publish&subscribe source
   par n:nat
   invokes service
   handledBy iseven(n) ⊃ service?
```

This connector is made explicit in the configuration as a mediator:

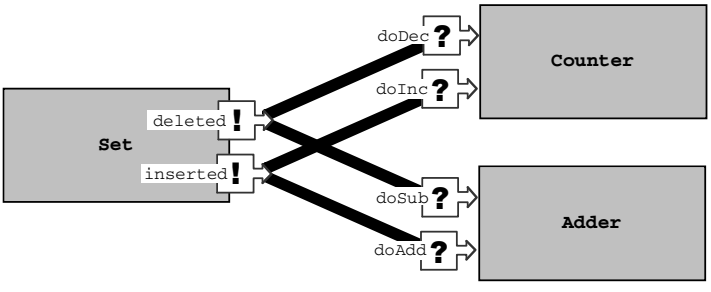


The same design approach can be applied to the addition of an Adder:

```
design Adder is
provide add
   par lm:nat
   assignsTo sum
   effects  sum'=sum+lm
provide sub
   par lm:nat
   assignsTo sum
   effects  sum'=sum-lm
```

```
store sum:nat
subscribe doAdd
   par which:nat
   invokes add
   handledBy add? ∧ which=add.lm
subscribe doSub
   par which:nat
   invokes sub
   handledBy sub? ∧ which=sub.lm
```

The required configuration is:



We abstain from translating the configuration to a categorical diagram. The colimit of that diagram returns the design *Set&Counter&Adder* discussed in section 2 and the morphisms that identify *Set, Adder* and *Counter* as components.

5 Combining Architectural Styles

Another advantage of the categorical formalisation of publish/subscribe is that it allows us to use this style in conjunction with other architectural modelling techniques, namely synchronous interactions as in CommUnity [6]. For instance, consider that we are now interested in restricting the insertion of elements in a set to keep the sum below a certain limit *LIM*. Changing the service *add* of *Adder* to

```
provide add
   par lm:nat
   assignsTo sum
   guardedBy sum+lm<LIM
   effects  sum'=sum+lm
```

does not solve the problem because *Adder* subscribes to *inserted* which is published after the element has been inserted in the set. What we need is to strengthen the enabling condition of *insert* in *Set* with *sum+lm<LIM* and ensure that *sum* is updated by *insert* and *delete*. However, to do so within ***DSGN*** we would have to redesign the whole system. Ideally, we would like to remain within the incremental approach through which we superpose separate components to induce required behaviour.

One possibility is to use action synchronisation and i/o communication as in CommUnity [6]. More precisely, the idea is to synchronise *Set* and *Adder* to ensure that *sum* is updated when insertions and deletions are made, and superpose a regulator to check the sum before allowing the insertion invocation to proceed.

Consider the synchronisation of *Set* and *Adder* first. In CommUnity, actions capture synchronisation sets of service invocations, something that is not intrinsic to implicit invocation as an architectural style and, therefore, cannot be expressed in the formalism presented in the previous sections. Our first step is to extend the notion of design with synchronisation constraints and communication channels.

Definition: *We call an extended signature $Q^{I,O}$ a signature Q together with two D-indexed families I and O of mutually disjoint finite sets (of input and output channels, respectively). An extended design over $Q^{I,O}$ is a tuple $<\eta,\rho,\gamma,\beta,\chi>$ where $<\eta,\rho,\gamma>$ is a design for Q in which I can be used in the languages of ρ and γ, and:*

- *β is a proposition establishing what observations of the local state (variables) are made available through the output channels.*
- *χ is a proposition in the language of services and their parameters establishing dependencies that need to be observed on execution.*

As an example, consider the following revision of *Set&Counter&Adder:*

```
design syncSet&Counter&Adder is
store elems: set(nat),
      value:nat, sum:nat
output mysum:nat
publish&subscribe inserted
   par which:nat
   invokes inc
   handledBy inc?
publish&subscribe deleted
   par which:nat
   invokes dec
   handledBy dec?
subscribe doInsert
   par which:nat
   invokes insert
   handledBy insert? ∧
     which=insert.lm
subscribe doDelete
   par which:nat
   invokes delete
   handledBy delete? ∧
       which=delete.lm
synchronise insert≡add ∧
       insert.lm=add.lm ∧
       sub≡delete ∧
       sub.lm=delete.lm
convey mysum=sum
```

```
provide insert
   par lm:nat
   assignsTo elems
   publishes inserted
   guardedBy lm∉elms ∧ lm+sum<LIM
   effects  elems'={lm}∪elems ∧
     inserted! ∧ inserted.which=lm
provide delete
   par lm:nat
   assignsTo elems
   publishes deleted
   guardedBy lm∈elms
   effects  elems'=elems\{lm} ∧
      deleted! ∧ deleted.which=lm
provide inc
   assignsTo value
   effects  value'=value+1
provide add
   par lm:nat
   assignsTo sum
   effects  sum'=sum+lm
provide sub
   par lm:nat
   assignsTo sum
   effects  sum'=sum-lm
provide dec
   assignsTo value
   effects  value'=value-1
```

Through *synchronise* we provide a proposition that defines the synchronisation sets of service activation that can be observed during execution. For instance, through $a \equiv b$, we can specify that two given services *a* and *b* are always activated simultaneously. Hence, in the example, *insert* and *add* are always performed synchronously.

Through *convey* we establish how the output channels relate to the state variables. In the example, we are just making the *sum* directly available to be read by the envi-

ronment through *mysum*. Notice that we have also strengthened the guard of *insert* with the condition *lm+sum<LIM*.

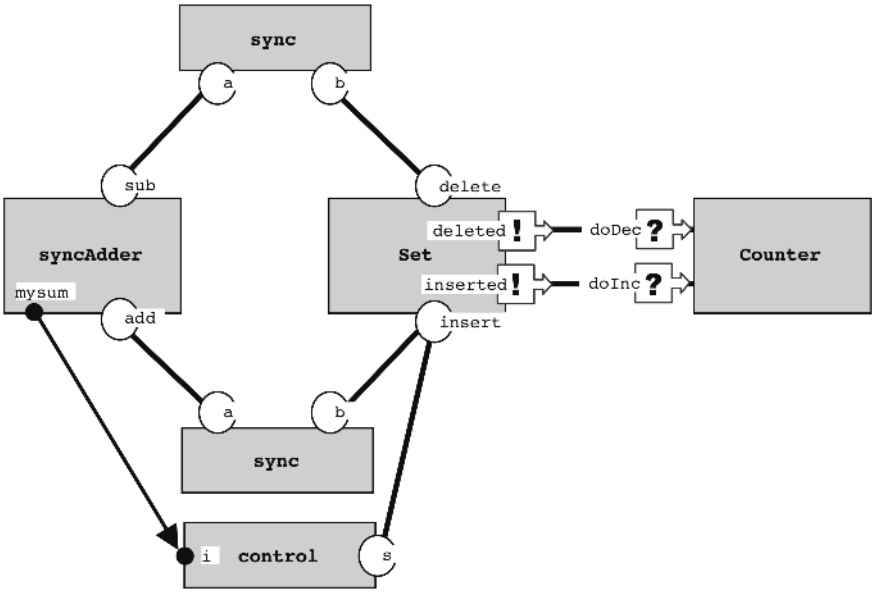
It remains to show how we can externalise the extension. The following design captures the synchronisation:

```
design sync is
synchronise a≡b
     ∧ a.p=b.p
provide a
   par p:nat
provide b
   par p:nat
```

For strengthening the guard of *insert* we need a component that reads the state of *Adder* to determine if *insert* can proceed:

```
design control is
input i:nat
provide s
   par n:nat
   guardedBy n+i<LIM
```

This leads us to the following configuration:



Notice that *syncAdder* is given by the following design:

```
design syncAdder is
provide add
   par lm:nat
   assignsTo sum
   effects  sum'=sum+lm
provide sub
   par lm:nat
   assignsTo sum
   effects  sum'=sum-lm
store sum:nat
output mysum:nat
convey mysum=sum
```

The proposed extension is supported by the following notion of morphism:

Definition: *A morphism* σ *between extended signatures* $<V_1,E_1,S_1,P_1,T_1,H_1,I_1,O_1>$ *and* $<V_2,E_2,S_2,P_2,T_2,H_2,I_2,O_2>$ *is a morphism between signatures* $<V_1,E_1,S_1,P_1,T_1,H_1>$ *and* $<V_2,E_2,S_2,P_2,T_2,H_2>$ *together with* $\sigma_{in}:I_1 \rightarrow I_2 \cup O_2$ *and* $\sigma_{out}:O_1 \rightarrow O_2$.

That is, as in CommUnity [6], input channels may become output channels of the system but not the other way around.

Definition: *A morphism between* $<\eta_1,\rho_1,\gamma_1,\beta_1,\chi_1>$ *and* $<\eta_2,\rho_2,\gamma_2,\beta_2,\chi_2>$ *is a morphism between* $<\eta_1,\rho_1,\gamma_1>$ *and* $<\eta_2,\rho_2,\gamma_2>$ *such that the observation and synchronisation dependencies are preserved:* $\Phi \vdash \beta_2 \supset \underline{\sigma}(\beta_1)$ *and* $\Phi \vdash \chi_2 \supset \underline{\sigma}(\chi_1)$.

Notice that this is an extension of the previous notion, i.e. morphisms between designs that do not involve communication channels and synchronisations are as before. Further details on this extension, including the way it relates to CommUnity, can be found in a companion paper.

6 Conclusions and Further Work

In this paper, we presented a formalisation of the architectural style known as "publish/subscribe" or "implicit invocation". Full details on the mathematics involved as well as the semantics of publication and notification can be found in a companion paper. This formalisation allowed us to further validate the approach to software architecture introduced in [7].

Other formal models [e.g., 4,9] exist that abstract away from concrete notions of event and related notification mechanisms. However, they address the computational aspects of the paradigm, which is necessary for supporting, for instance, several forms of analysis. Our work addresses primarily the architectural properties of the paradigm, i.e. what concerns the way connectors can be defined and superposed over components to coordinate their interactions.

In particular, our formalisation allowed us to characterise key structural properties of the architectural style in what concerns the externalisation of bindings and mediators previously claimed in papers like [14]. These properties derive from the fact that the (forgetful) functor that maps the category of designs to that of signatures has the strong structural property of being coordinated, as explained in [6]. We should stress that these structural properties result from the nature of the morphisms that we defined in section 3, which may leave some readers who are not aware of the complexity of the mathematics involved somewhat disappointed and wishing to have seen more results… It is true that, in this paper, we have "only" defined a category and a (forgetful) functor, but both satisfy very strong properties that can be used for further exploring implicit invocation as an architectural style.

Furthermore, the proposed categorical semantics allows us to investigate how this style can be used in conjunction with other architectural techniques. In section 5, we addressed the way implicit invocation can be used together with synchronous forms of interconnection as previously formalised through the language CommUnity [6]. CommUnity itself has been extended in other ways, for instance with primitives that capture distribution and mobility [8] as well as context awareness [11]. Further work

is going on towards exploiting this categorical framework to support the integration of several architectural styles.

Acknowledgements

This work was partially supported through the IST-2005-16004 Integrated Project *SENSORIA: Software Engineering for Service-Oriented Overlay Computers*. We would like to thank the reviewers for having provided so much feedback.

References

1. J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, M. Spiteri (2000) Generic support for distributed applications. *IEEE Computer* 33(3):68–76
2. J. Bradbury, J. Dingel (2003) Evaluating and improving the automatic analysis of implicit invocation systems. In: *ESEC/FSE'03*. ACM Press, pp 78–87
3. A. Carzaniga, D. Rosenblum, A. Wolf (2001) Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems* 19:283–331
4. J. Dingel, D. Garlan, S. Jha, D. Notkin (1998) Towards a formal treatment of implicit invocation. *Formal Aspects of Computing* 10:193–213
5. P. Eugster, P. Felber, R. Guerraoui, A-M. Kermarrec (2003) The many faces of publish/subscribe. *ACM Computing Surveys* 35(2):114–131
6. J. L. Fiadeiro (2004) *Categories for Software Engineering*. Springer, Berlin Heidelberg New York
7. J. L. Fiadeiro, A. Lopes (1997) Semantics of architectural connectors. In: M. Bidoit, M. Dauchet (eds) *TAPSOFT: Theory and Practice of Software Development*. *LNCS, vol 1214*. Springer, Berlin Heidelberg New York, pp 505–519
8. J. L. Fiadeiro, A. Lopes (2004) CommUnity on the move: architectures for distribution and mobility. In: M. Bonsangue et al (eds) *Formal Methods for Objects and Components. LNCS, vol 3188*. Springer, Berlin Heidelberg New York, pp 177–196
9. D. Garlan, S. Khersonsky, J. S. Kim (2003) Model checking publish-subscribe systems. In: T. Ball, S. Rajamani (eds) *Model Checking Software. LNCS, vol 2648*. Springer, Berlin Heidelberg New York, pp 166–180
10. D. Garlan, D. Notkin (1991) Formalizing design spaces: Implicit invocation mechanisms. In: S. Prehn, W. J. Toetenel (eds) *VDM'91: Formal Software Development Methods*. *LNCS, vol 551*. Springer, Berlin Heidelberg New York, pp 31–44
11. A. Lopes, J. L. Fiadeiro (2005) Context-awareness in software architectures. In: R. Morrison, F. Oquendo (eds) *Software Architecture*. *LNCS, vol 3527*, Springer, Berlin Heidelberg New York, pp 146–161
12. R. Meier, V. Cahill (2002) Taxonomy of distributed event-based programming systems. In: *Proceedings of the International Workshop on Distributed Event-Based Systems*. IEEE Computer Society, Silver Spring, MD, pp 585–588
13. D. Notkin, D. Garlan, W. Griswold, K. Sullivan (1993) Adding implicit invocation to languages: three approaches. In: S. Nishio, A. Yonezawa (eds) *Object Technologies for Advanced Software*. *LNCS, vol. 742*, Springer, Berlin Heidelberg New York, pp 489–510
14. K. Sullivan, D. Notkin (1992) Reconciling environment integration and software evolution. *ACM TOSEM* 1(3):229–268

.

Engineering Self-protection for Autonomous Systems

Manuel Koch and Karl Pauls

Freie Universität Berlin,
Institut für Informatik,
Takustr. 9, D–14195 Berlin, Germany
{mkoch, pauls}@inf.fu-berlin.de

Abstract. Security violations occur in systems even if security design is carried out or security tools are deployed. Social engineering attacks, vulnerabilities that can not be captured in the relatively abstract design model (as buffer-overflows), or unclear security requirements are only some examples of such unpredictable or unexpected vulnerabilities. One of the aims of autonomous systems is to react to these unexpected events through the system itself. Subsequently, this goal demands further research about how such behavior can be designed and sufficiently supported throughout the software development process. We present an approach to engineer self-protection rules for autonomous systems that is integrated into a model-driven software engineering process and provides concepts to formally verify that a given intrusion response model satisfies certain security requirements.

1 Introduction

Model building as a means of producing appropriated documentation, providing specifications, and code generation is a standard practice in software engineering. Security, as an integral part of any modern software system that is not used in completely trusted environments, demands systematic support for software engineers who need to produce secure software. Considering security aspects throughout the entire software development process (and not during the requirements analysis and system integration phases only) by explicitly integrating security into the design models can aid in detecting and removing potential security breaches. Furthermore, model-centric and generative approaches, as the concept of Model Driven Architecture (MDA) [5], have led to advancing support for software engineers. Most noticeably, the *Model Driven Security* approach (see [3]) proves to not only define access-control languages (in this case *SecureUML*) but to provide a basis for refinement down to code as well. Access control is concerned with preventing unauthorized accesses to shared resources. Which accesses are authorized depends on specific security requirements and has to be specified in access control policies.

A model is an abstract part of the real world which contains the aspects relevant to the developer. It does not deal with any environment interaction. In

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 33–47, 2006.

particular from the viewpoint of security, not all possible attacks can be considered in a model. This may be because of the abstraction level of the model (e.g. buffer overflows cannot be seen in class diagrams) or because of environment parts that are not considered in a model (e.g. social engineering) or other reasons. Therefore, crucial functions must be monitored. Concerning security, accesses to crucial data must be logged to see if unauthorized access occur despite a given access control policy (e.g. by buffer overflows). In very sensitive environments, intrusion detection systems (IDS) [19] and security engineering [1], in practice, often need to be combined so that the occurrence of interactions and system states not covered by the normal use cases raises an alarm in an IDS. Consequently, the question of how to identify nonconforming or malicious behavior and how to react to discovered security breaches arises.

Ever since IBM's "call to arms" [6], called autonomous computing, research increasingly focuses more effort on self-adapting, self-protecting, and self-healing systems (i.e., autonoums systems) [18]. Monitoring the system is a necessary aspect in autonomous systems to detect system or environment changes and possibly to react on these changes if necessary. Self-repairing/Self-protecting systems are an ambitious goal whose realization concerns many aspects of software engineering (for example Baresi et. al considered self-healing in service-oriented systems in regard to dynamic binding of services in [2]). We consider in this paper the self-protection of the system in the case of successful security attacks.

We take up the challenge of providing sufficient support to design and realize self-protecting system. We present an approach to engineer self-protection rules for autonomous systems, integrated into a model-driven development approach, and capable to generate self-protecting access control aspects for XACML based infrastructures. Furthermore, we provide concepts based on graph transformations [17] to formally verify that a given intrusion response model satisfies certain security requirements.

The remainder of this article is organized as follows. We give next a description of the model-driven development approach and the underlying concepts of our access control model together with an operational semantic. Section 3 concerns the specification of protection rules. Section 4 presents the concepts to verify the satisfaction of security requirements. Finally, we present related work and conclusion.

2 Model-Driven Development

We first present the integration of our approach into a model-driven development approach namely, *Model Driven Security* and its underlying access control model *SecureUML*. It concerns the development of XACML [11] based access control policies and access control properties following an attribute based access control approach which can be described by a mapping of the modeling elements of *SecureUML* to XACML policies. Afterwards, an additional operational semantic of the entity operations is given that serves as a starting point for our self-protecting rules requirement analysis.

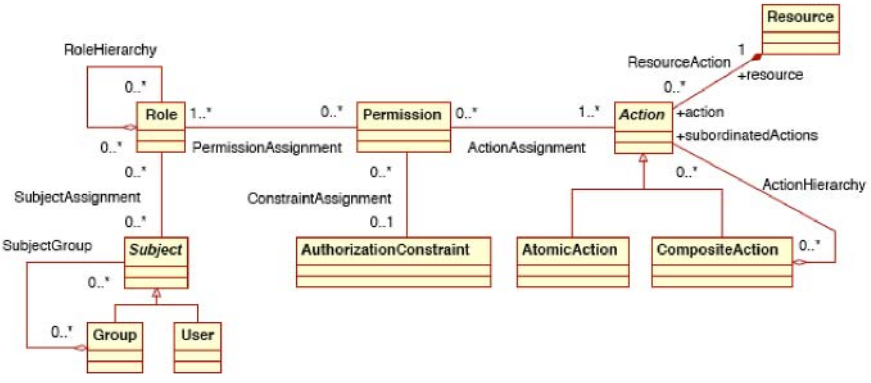


Fig. 1. RBAC metamodel

As already stated above, we build up on the *SecureUML* metamodel presented in [3]. Figure 1 presents the role based access control (RBAC) based metamodel that defines the abstract syntax for *SecureUML*. Due to space constraints we can not go in too much detail about the concrete syntax and semantics but refer the interested reader to [3]. On the left-hand side of the diagram RBAC is formalized. Users can be assigned to Groups. On the other side of the model, permissions, which can be assigned to roles, are used to model the ability to carry out actions on resources on behalf of a calling subject that is in a certain role. Authorization constrains can be used to constrain that certain permissions only hold in certain system states.

To this end, we require that subjects have or provide certain properties (credentials) to be assigned to roles. For example, a user name, a counter for logins, a printer quota, location in mobile scenarios etc. This approach is called attribute-based access control (ABAC) [16]. The main idea of ABAC is to dynamically define the authorization of subjects based on current property values of the calling subjects and their targeted resources, respectively. In addition to the relatively static defined roles, this attributes can be highly dynamic therefore, provide a way to capture the needs of e-commerce as well as enterprise and e-government applications in the internet ranging all the way to ubiquitous computing.

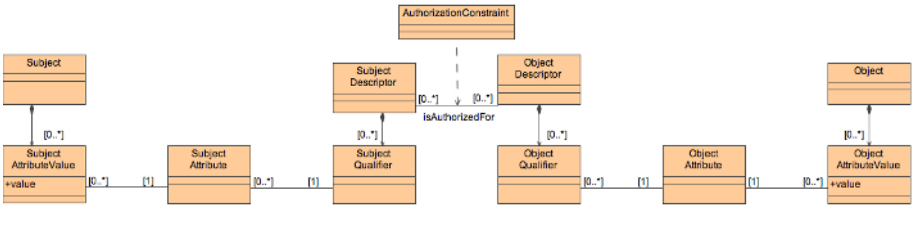


Fig. 2. ABAC model

Figure 2 shows our base model of the applied ABAC approach. In general, we enhance the aforementioned metamodel in Fig. 1 by this explicit specification of subject and object attributes.

2.1 Generation of XACML Based Access Control Policies

Model Driven Security gives a means to integrate access control concerns into a model and subsequently generate code out of this model elements. In [3] concrete mappings for EJB and .Net are given. In addition, we focus on the generation of XACML based access control policies since, policy based infrastructures are more flexible and, more specific, are able to provide better support for policy changes and management then the standard security architectures of todays enterprise systems. To this end, we define and implemented a UML profile that enables to generate XACML policies from security models (i.e., models that are build using our ABAC enhanced *SecureUML* modeling elements).

2.2 Example and Operational Semantics of the Protection Model

In addition to the SecureUML concrete semantic, our protection model contains an operation semantic of the entity operations. As a running example, we will consider developing a simplified version of a system for administrating calendars. Figure 3 shows the simple interface of the calendar application that basically, allows to create a calendar and subsequently, create, update, delete, and read entries (i.e., appointments).

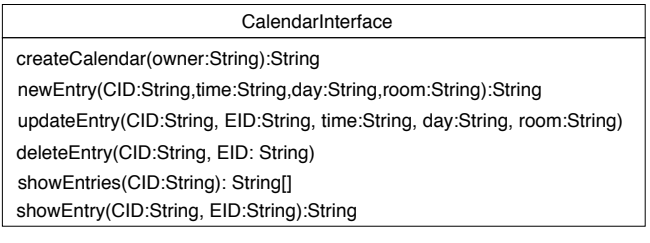


Fig. 3. Interface of the Calendar Application

Considering this simple interface, one may want to enforce some basic integrity properties like, for example that the creator of a calendar becomes its owner hence, is the only one that is allowed to delete entries. Arbitrary users are allowed to read the entries of any calendar but modification is up to the owner of the calendar or a substitute like a secretary. At the very least a secretary should be able to make and manage appointments (i.e., create and update entries in the calendar). Lets assume that the deletion of an entry is restricted to the owner of the calendar only. These security requirements are implemented in the model in Fig. 4 which is an instance of the *SecureUML* metamodel. We have three roles

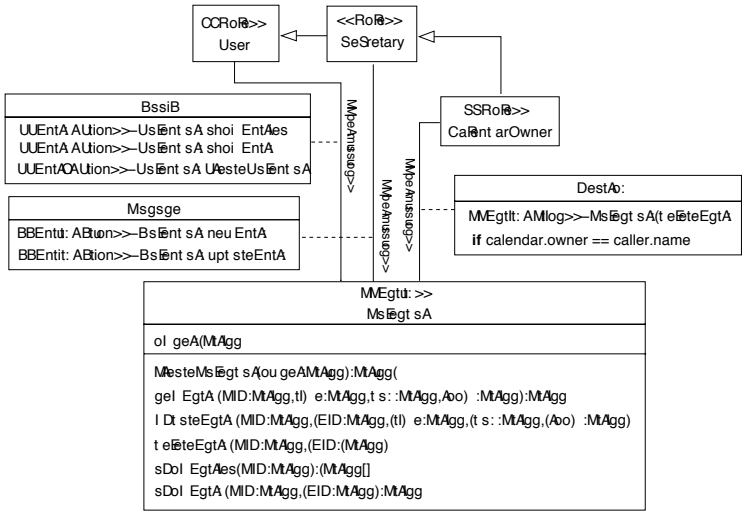


Fig. 4. security model

and three permissions. The permission *Basic* allows a subject in role User to read calendar entries and to create calendars. The permission *Manage* allows subjects in role Secretary to modify a calendar and permission *Destroy* allows subjects in role CalendarOwner to delete calendar entries.

Once the security model is accomplished, the operational semantics of the entity operations have to be specified. We define the notion of a *protection model* as follows. A *protection model* is a pair of a security model (as described above) and a set of transformation rules [17]. Considering our example we write, the security model of figure 4 as M and the protection model as a pair $(M, ORules)$. Figure 5 gives the transformation rules for the creation of a calendar and the deletion of entries in the calendar via the *create*() and *deleteEntry*() methods of the *Calendar* entity. A transformation rule consists of two object diagrams. The diagram on the left-hand side of a rule models the precondition to apply the rule. The object diagram on the right-hand side models the transformed object state. The left-hand side of rule *create(x)* requires a subject with name x in a role and this role must have a permission with entity action create on the calendar entity. If this object structure can be found in a system state, a new calendar object for the subject with name x is created. The left-hand side of rule *deleteEntry* requires a subject and a connected calendar object. The subject must be in a role which has a permission for entity action *deleteEntry*. The condition *calendar.owner* $== x$ enforces that the rule can be only applied if the subject is the owner of the calendar. The effect of the rule is the deletion of an entry of the calendar object. Transformation rules can be mapped to graph rules to give them a formal semantics [8, 17].

Ultimately, the transformation rules capture the aforementioned attributes of our subject and object descriptors and more importantly, their changes during the state changes of the system. Since we assume that state changes are triggered

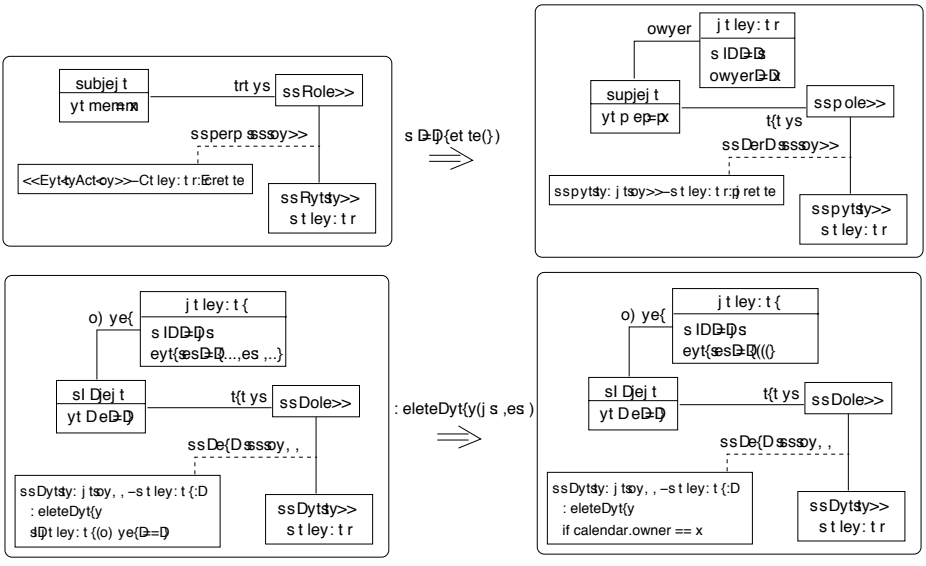


Fig. 5. Operational semantics for calendar operations create() and deleteEntry()

by method calls our notion is sufficient to model their impact on the protection model. In figure 5 for example, the create(x) call to the calendar interface sets the owner attribute of the calendar to the name of the caller. In consequence, we can build up on this attribute to restrict the deletion of an entry to the actual creator of the calendar (i.e., its owner) by comparing the name attribute of the caller with the owner attribute of a targeted calendar.

3 Specification of Self-protection Rules

The analysis of security requirements for a software system is a difficult design task and recent research focuses on developing models and concepts to elicit, analyze and document security requirements [9, 10]. We assume in this article, that security requirements are documented and a risk assessment has given them a priority. The following list shows the examples used in the remainder of this article.

Security Requirement C1: Prevent that a calendar has more than one owner.
Security Requirement C2: Prevent that a user is logged into the system more than once.
Security Requirement C3: Prevent denial of service attacks by creating more than n calendars.

These in natural language formulated requirements can be specified in semi-formal or formal constraint languages (e.g., OCL[13]) and models can be checked

if they satisfy these requirements. The focus of this article, however, is not this static check of design models, but we are interested in the security vulnerabilities that are detected during run-time even if the model is previously checked to be secure.

Therefore, a crucial component of autonomous systems is a monitor which observes the system states during run-time in order to detect constraint violations. When the monitor detects an insecure system state the system should react by protection means (in the case of an autonomous system the system reacts autonomously). One possible solution to protect the system would be a system shutdown or to disconnect the whole system. This solution, however, is quite rigid and restricts the availability of the system often more as necessary. If the originator of a security violation can be determined it would possibly be enough to eliminate this user from the system. If the originator is unknown, it is sufficient to disconnect the attacked subsystem or restrict its functionality, so that the remaining system can continue working in a restricted mode.

Before we present an approach to specify a more fine-grained protection, we differentiate between *self-protection* and *self-repairing* of the system. Self-protection changes the security model by revoking permissions as far as necessary so that an intruder cannot do any harm with the acquired authorization, but the system state remains unchanged. Self-repairing, on the other hand, transforms the insecure system state into a secure state and lets the security model unchanged. Self-repairing, i.e. an automatic modification of the system state without any interaction with an administrator, is often difficult to implement. Consider as an example a violation of the requirement $C1$ from above, i.e., the monitor detects two calendar owners for a calendar. Should we revoke both owners from the calendar (but then we have calendars without owners) or should we only revoke one calendar owner (but which one, which owner is the "real" owner)? In the case of a violation of requirement $C3$, calendars must be removed to reach the maximum boundary of allowed calendars. But, which calendar should be removed?

We focus next on the specification of self-protection. For each security requirement, a set of protection rules models the reaction of the system to the violation of the security requirement and transforms the protection model. The transformation should restrict the model as far as necessary and should allow system availability as far as possible. The protection rules are developed in two steps:

1. *Specify the response requirement.* A response requirement for a security requirement specifies the system functionality which must be restricted in the case of a security requirement violation.
2. *Specify the protection sets for the response requirement.* A protection set contains a set of transformation rules to restrict the security model. The rules of a protection set for a response requirement shall satisfy the response requirement.

3.1 Development of the Intrusion Response

To support the system designer in finding the appropriate response requirements, we suggest an approach which is driven by the UML models, since static diagrams (as class diagrams) contain the elements that should be protected (in our example the calendars and their entries), the behavior diagrams (as sequence diagrams) show how the protected elements are accessed. Therefore, the designer decides on the basis of these UML models the measures to do in the violation response. Consider the security requirement $C1$ for at most one calendar owner as an example. The designer has the class diagram in Fig. 4 and the sequence diagrams in Fig. 6 as documentation and assumes now that requirement $C1$ can be violated by a security vulnerability so that an attacker can become owner of calendars of other persons. When the designer considers the sequence diagrams

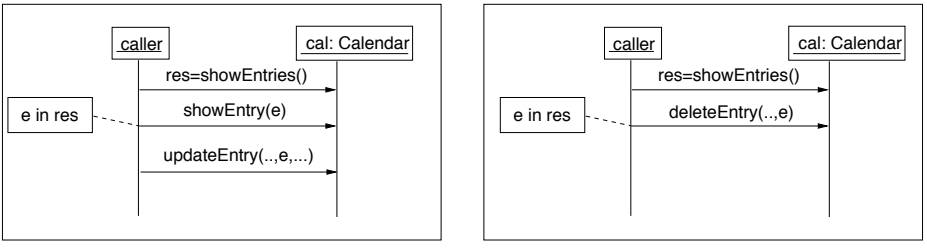


Fig. 6. Sequence diagrams for updating a calendar entry and for deleting an entry

in Fig. 6, (s)he realizes that the attacker can call the operations *showEntries()*, *showEntry()*, *updateEntry()* and *deleteEntry()*. While an unauthorized call of the operations *showEntries()* and *showEntry()* appears to be an acceptable (i.e., it does not concern integrity) risk, compared to disabling read access for all (including the trustworthy) users, an unauthorized call of the operations *updateEntry()* and *deleteEntry()* cannot be tolerated. Therefore, the designer adds *updateEntry()* and *deleteEntry()* for calendar owner to the response requirement, i.e., both operations must not be called by calendar owners when requirement $C1$ is violated. Analog, the response requirements for the other security requirements are specified driven by the UML diagrams.

The list below shows the response requirements of our calendar example. A response requirement is a set of pairs $(Caller, Operations)$ consisting of a list of callers $Caller$ who are not allowed to call the operations in the set $Operations$ in the case of the corresponding security violation.

$$Response(C1)=\{(CalendarOwner,\{deleteEntry(), newEntry(), updateEnty()\})\}$$
$$Response(C2)=\{(User,\{deleteEntry(), newEntry(), updateEnty(), createCalendar()\})\}$$
$$Response(C3)=\{(User,\{createCalendar()\})\}$$

3.2 Specification of Self-protection

A first idea to satisfy the response requirement would be to generally disallow callers to call the operations in the response requirement. Since the response requirements are connected to certain callers, however, this general prohibition is too strong. To restrict the operations to certain callers requires additional operation conditions. Since operations are implemented and a code change during run-time is not desirable, operations cannot be modified with respect to the response requirement. Therefore, the security model must be modified so that only the specific callers are affected. A change of the security model can be done during run-time and is immediately enforced by the XACML infrastructure [11].

The security model transformation is specified by a set of graph rules (Fig. 7 shows a part of the graph rules for the calendar example). The *protection sets* for a response requirement contain a subset of the protection rules. The protection set $Protect(C1)$ to satisfy the response requirement $Response(C1)$ removes all permissions from role calendar owner and adds a permission to calendar owner to read calendars. The protection set $Protect(C2)$ removes all permissions to modify a calendar and introduces a restricted basic permission which allows the user to read the calendars only. The protection set $Protect(C3)$ removes the permission to create calendars by adding a restricted basic view.

$Protect(C1)$={remove destroy(CalendarOwner), remove inheritance, add basic(CalendarOwner)}.
$Protect(C2)$={remove destroy(CalendarOwner), remove manage(Sectretary), replace basic, add basicrestricted(User)}.
$Protect(C3)$={replace basic, add basicrestricted(User)}.

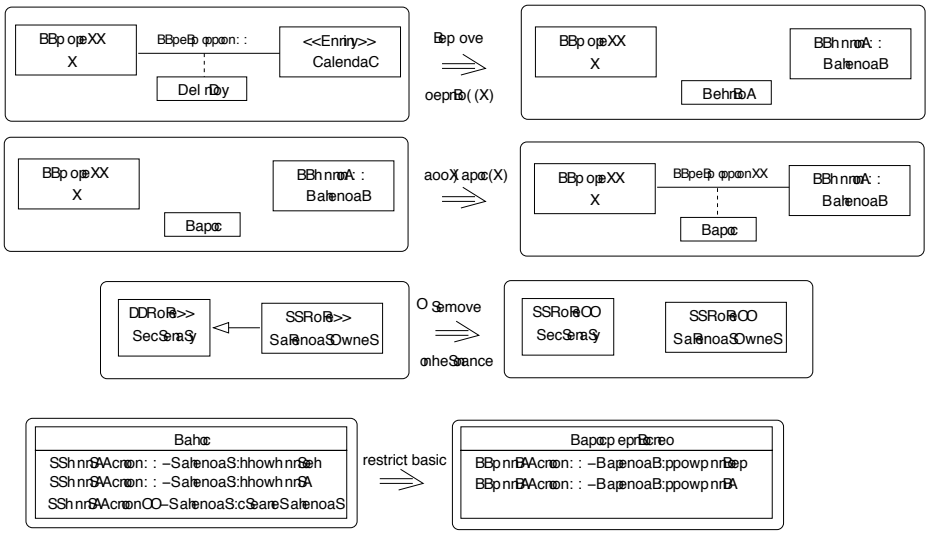


Fig. 7. The protection rules

When the system monitor detects a violation of a security requirement (e.g., $C1$) the rules in the protection sets are executed (e.g., $Protect(C1)$) to ensure the response requirement (e.g., $Response(C1)$). Figure 8 shows the results of applying the protection sets $Protect(C1)$, $Protect(C2)$ and $Protect(C3)$, respectively, to the security model in Fig. 4. When several security requirements

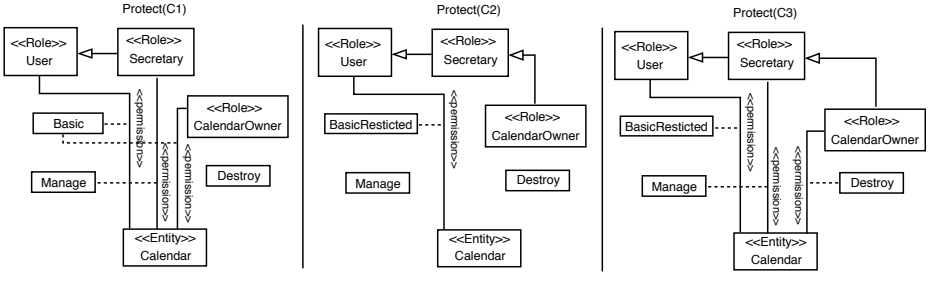


Fig. 8. The security models after execution of $Protect(C1)$, $Protect(C2)$ and $Protect(C3)$

are violated at the same time, several protection sets are applied. The response requirement for two security requirements $C1$ and $C2$ is $Response(C1 + C2) = Response(C1) \cup Response(C2)$. We define the protection set as $Protect(C1 + C2) = Protect(C1) \cup Protect(C2)$.

One could argue that, instead of specifying the protection rules, it would be easier to specify immediately the restricted security models. Since there must be a security model for each combination of violated security requirements, one has to specify $2^n - 1$ security models in the case of n requirements. Therefore, for a bigger n it is certainly more convenient to specify n rule sets which ensure that each constructed security model is consistent. The next section concerns this consistence statement.

4 Protection Satisfaction

A protection set contains rules which modify the security model in the case of unexpected security requirement violations. By now, there is no restriction on the ordering in which the rules of a protection set must be applied and one can wonder if any order results in the same security model or if the ordering is relevant. A second question is whether the security model constructed by the rules of a protection set satisfy the response requirement. Therefore, this section concerns the following questions.

1. Does the rule application ordering influence the final security model?
2. Does a protection set satisfies a response requirement?

4.1 Dependencies Between Protection Rules

If a protection set becomes necessary to protect the system against a security violation, each rule in the protection set is applied once. Since the ordering is by default unrestricted, the following problems may occur (see also independence of graph transformations in [17]).

Problem 1: Assume rules p_1 and p_2 which are both applicable to the security model M, but rule p_1 deletes elements required by p_2, so that an application of p_1 prevents the applicability of p_2. Dependent on the rule ordering, two different security models M' are generated and, therefore, the two rules are in conflict. To detect these conflicts, *critical pair analysis* of graph rules [15, 4] can be used. The critical pairs for two rules are constructed by overlapping the rule left-hand sides in all possible ways, such that the intersection contains at least one deleted element. In this way, critical pairs show all the potential conflicts between the rules in a minimal context. Each actual conflict in a bigger context will be represented by one of the critical pairs.

There is tool support for generating the critical pairs for rules implemented in the AGG tool [20]. Figure 9 shows the result of the critical pair analysis for our example rules. The tool detects a critical pair for rules *replaceBasic* and *addBasic(CO)*. This bases on the fact, that rule *replaceBasic* deletes the permission *Basic* which in turn is required in the left-hand side of rule *addBasic(CO)*. Therefore, applying first rule *replaceBasic* prevents the application of rule *addBasic(CO)*.

Critical Pairs

first \ second	1: removeDestro...	2: addBasic(CO)	3: removeInherit...	4: replaceBasic	5: removeManag...	6: addBasicRestri...
1: removeDestroy(CO)	0	0	0	0	0	0
2: addBasic(CO)	0	0	0	1	0	0
3: removeInheritance	0	0	0	0	0	0
4: replaceBasic	0	1	0	0	0	0
5: removeManage(Sec)	0	0	0	0	0	0
6: addBasicRestricted(U)	0	0	0	0	0	0

Fig. 9. Computation of critical pairs by AGG

After the computation of critical pairs for a protection set, the rule set can be divided into conflict free rules (rules which do not have critical pairs) and conflicting rules. Conflict free rules can be applied in any order (from the viewpoint of problem 1, we will see next another problem which additionally influences the rule application order) and the security engineer can use them in any combination in the protection sets to get a single final security model.

Conflicting rules should not be used together in a protection set or the security engineer must specify the desired application order. In our example, (s)he could specify that *addBasic(CO)* must always be applied before *replaceBasic*.

Problem 2: Assume now rules p_1 and p_2, so that p_1 creates elements required by p_2. Then, rule p_2 can be applied only after p_1, but not before (see also sequential dependence in [17]). These conflicts are called *sequential dependence* conflicts and can be detected by considering the overlaps of the right-hand side of rule p_1 and the left-hand side of rule p_2. There is no conflict if the left-hand side of p_2 does not require elements which are generated by p_1. Otherwise, there is a conflict. If we investigate the rules of our example, we see that the rule *addBasicRestricted(U)* is sequential dependent of *replaceBasic* since *replaceBasic* generates the permission *BasicRestricted* required by rule *addBasicRestricted(U)*. All other rules are not sequential dependent.

Analog to critical pair analysis, sequential dependencies between protection rules can be automatically detected and presented to the security engineer who uses this information in the specification of the protection sets.

4.2 Satisfaction of the Response Requirement

Applying a protection set to a protection model $(M, ORules)$ results in a new protection model $(M', ORules)$ in which the security model is changed (from M to M'). The operation rules $ORules$ remain unchanged under this transformation. The permission or denial of operation accesses must now be checked with respect to the new security model M'.

A protection set *satisfies* a response requirement Res, if for any pair $(Caller, Operations)$ in Res, none of the transformation rules for an operation in $Operations$ can be applied to $Caller$ in the changed security model M'. This satisfaction can be checked by considering the left-hand sides of the transformation rules in the response requirement Res and the new model M'. If the security relevant part of the left-hand side (which consists of all elements with stereotype <<Role>>, <<EntityAction>>, <<Permission>> and <<Entity>>) of a rule p in Res can be embedded into the security model M' then one can construct a state for M' to which p can be applied (mainly the left-hand side itself). Therefore, the response requirement is not satisfied. On the other hand, if the security relevant part of the rule cannot be embedded into the security model, this part can neither be embedded into a state for M'. This means that the rule is never applicable and the response requirement is satisfied.

Consider as an example the protection set $Protect(C1)$ for the security requirement $C1$. The modified security model M' is shown in Fig. 8 on the left-hand side. The response requirement $Response(C1)$ forbids a calendar owner for example to call the operation *deleteEntry()*. The security relevant part of the left-hand side of the transformation rule for *deleteEntry()* (bottom of Fig. 5) cannot be embedded into the security model M', since the rule requires a role which has a permission on the calendar entity, and the permission contains an entity action *deleteEntry*. In the security model in Fig. 8, however, no role is

connected to the permission *Destroy* (the only permission with entity action *deleteEntry*). Therefore, the rule for *deleteEntry()* cannot be applied to any system state corresponding to the security model M'.

4.3 Benefits for the Security Engineer

At the end of this section, we summarize how the answers of the two questions in the beginning of this section can support the security engineer in designing a self-protection system.

1. *Does the rule application ordering influence the final security model?* Paragraph 4.1 has shown that different rule application orderings may lead to different security models. The security engineer, however, can use critical pair analysis (supported by the AGG tool) and sequential dependence analysis to compute the conflicting rules. Considering these results in the engineering process of the protection sets allows the security engineer to get a deterministic behavior of the protection set response.
2. *Does a protection set satisfies a response requirement?* Paragraph 4.2 has presented a way to check whether the rules in a protection set satisfy a response requirement by considering the left-hand side of the transformation rules which specify the operations in the response requirement. If the designer detects rules in a protection set which does not satisfy a response requirement, (s)he must change the protection set or the protection rules until all response requirements are satisfied.

5 Related Work

Our approach uses the security engineering model presented in [3] for which tool support is given by an integration of the SecureUML metamodel into the ArcStyler tool [12]. The analysis stage of the software process, however, is not considered but the process starts with the design models. Jürjens presents in [7] the integration of security into the UML. He shows how to model several security aspects by UML model elements as, for example, stereotypes or tagged values. His approach is more general than ours since it is not restricted to access control but considers, for example, also security protocols. In [14, 21] approaches to design intrusion detection systems are presented. The design, however, focusses on the detection of attackers, less on the design of the response of an attack. Baresi et. al considered self-healing in service-oriented systems in regard to dynamic binding of services in [2].

6 Conclusion and Future Work

We presented a model-driven approach to engineer self-protection for autonomous systems. The approach is integrated into model driven security *SecureUML* for modeling access control and supports the system designer in engineering self-protection rules to react to unexpected security vulnerabilities.

Self-protection is specified by a set of transformation rules which restrict the security model. A graph-based semantics for the transformation rules allows us to verify that security requirements are satisfied by the specified self-protection rules.

We target an XACML based infrastructure which enforces the security model transformation that result by the self-protection sets. Furthermore, the XACML policies shall be generated from the models and protection rules. Another point of future work is the specification of the cancelation of self-protection restrictions. In other words, if the reason that causes the insecure state is eliminated, we have rules which transform the restricted model back into an unrestricted safe system.

References

1. G.-J. Ahn and R. Sandhu. Role-Based Authorization Constraints Specification. *ACM Transactions on Information and System Security*, 3(4):207–226, Nov. 200.
2. L. Baresi, C. Ghezzi, and S. Guinea. Towards Self-healing Compositions of Services. In *Proc. of PRISE'04, First Conference on PRInciples of Software Engineering*, pages 11–20, 2004.
3. D. Basin, J. Doser, and T. Lodderstedt. Model Driven Security: from UML Models to Access Control Infrastructures. *Journal of ACM Transactions on Software Engineering and Methodology*, 2005.
4. H. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graph transformation. In F. Parisi-Presicce, P. Bottoni, and G. Engels, editors, *Proc. 2nd Int. Conf. on Graph Transformation*, number 3256 in LNCS, pages 161–177. Springer, 2004.
5. D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing.* John Wiley and Sons., 2003.
6. P. Horn. Autonomic computing: IBM perspective on the state of information technology. Technical report, IBM T.J. Watson Labs, October 2001.
7. Jan Jürjens. *Secure Systems Development with UML.* Springer, 2005.
8. M. Koch and F. Parisi-Presicce. Access Control Policy Specification in UML. In *Proc. of UML2002 Workshop on Critical Systems Development with UML*, number TUM-I0208, pages 63–78. Technical University of Munich, September 2002.
9. M. Koch and K. Pauls. Generation of Role-based Access Control Requirements from UML Diagrams. In *Proc. of SREIS 2005, Symposium on Requirements Engineering for Information Security*, 2005.
10. N.R. Mead and T. Stehney. Security Quality Requirements Engineering (SQUARE) Methodology). In *Proc. of Software Engineering for Secure Systems (SESS05)*, 2005.
11. OASIS. *XACML 1.1 Specification*, August 2003.
12. Interactive Objects. Arcstyler, 2005. `www.io-software.com`.
13. OMG. *OCL 2.0 Specification, Version 2.0.* OMG, 2005.
14. M. M. Pillai, Jan H. P. Eloff, and H. S. Venter. An approach to implement a network intrusion detection system using genetic algorithms. In *SAICSIT '04: Proceedings of the 2004 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, pages 221–221, , Republic of South Africa, 2004. South African Institute for Computer Scientists and Information Technologists.

15. D. Plump. Hypergraph Rewriting: Critical Pairs and Undecidability of Confluence. In M. Sleep, M. Plasmeijer, and M.C. van Eekelen, editors, *Term Graph Rewriting*, pages 201–214. Wiley, 1993.
16. T. Priebe, W. Dobmeier, B. Muschall, and G. Pernul. ABAC – Ein Referenzmodell für attributbasierte Zugriffskontrolle. In *Proc. of Sicherheit 2005*, pages 285–296. Lecture Notes in Informatics GI–Edition, 2005.
17. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.
18. R. Sterritt. Autonomic computing. *Innovations in Systems and Software Engineering - A NASA Journal*, 1(1), 2005.
19. M. Stillerman, C. Marceau, and M. Stillman. Intrusion Detection for Distributed Applications. *Communications of the ACM*, 42(7):62–69, July 1999.
20. G. Taentzer, C. Ermel, and M. Rudolf. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter The AGG Approach: Language and Tool Environment. World Scientific, 1999.
21. Giovanni Vigna, Fredrik Valeur, and Richard A. Kemmerer. Designing and implementing a family of intrusion detection systems. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 88–97, New York, NY, USA, 2003. ACM Press.

A Graph-Based Approach to Transform XML Documents

Gabriele Taentzer[1] and Giovanni Toffetti Carughi[2]

[1] Technische Universität Berlin, Germany
gabi@cs.tu-berlin.de
[2] Politecnico di Milano, Italy
toffetti@elet.polimi.it

Abstract. As XML diffusion keeps increasing, it is today common practice for most developers to deal with XML parsing and transformation. XML is used as format to e.g. render data, query documents, deal with Web services, generate code from a model or perform model transformation. Nowadays XSLT is the most common language for XML transformation. But, although meant to be simple, coding in XSLT can become quite a challenge, if the coding approach does not only depend on the structure of the source document, but the order of template application is also dictated by target document structure. This is the case especially when dealing with transformations between visual models. We propose to use a graph-based approach to simplify the transformation definition process where graphs representing documents are transformed in a rule-based manner, as in XSLT. The differences to XSLT are mainly that rules can be developed visually, are more abstract (since the order of execution does not depend on the target document), IDREFs are dealt with much more naturally, and due to typed transformations, the output document is guaranteed to be valid with respect to the target schema. Moreover, graph-based transformation definitions can be automatically reversed in most cases. This is especially useful in model transformation (e.g. in OMG's MDA approach).

1 Introduction

When XML (Extensible Markup Language) [14] was being developed, the proposing working group at W3C had clear design goals in mind: they wanted to come up with a language which was at the same time formal, concise, easy to process for applications and to read and write for human beings. Today XML is used in virtually any IT domain as the most natural form to represent structured or (especially) semi-structured data. This includes usage of XML to store information, serialize models, communicate over the Internet, etc. As a consequence of this diffusion, it is common practice for most of today's programmers to deal with XML parsing and transformation, be it to render data, query documents, deal with web services, generate code from a model or perform model transformation.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 48–62, 2006.

XSLT (the Extensible Stylesheet Language Transformations [16]) is the language proposed by the W3C to deal with XML document transformation. Although developed to enable most IT developers to easily specify transformations, there are cases in which writing XSLT can be quite hard. The reasons are that, especially when dealing with model to model transformation, the coding approach does not only depend on the structure of the source document, but the order of template application is also dictated by target document structure. In addition to this, extensive use of IDREFs (i.e. references to other elements) can force developers to complicated composition of recursion, variables or keys to hop around the XML tree representation looking for some element.

We propose to use a graph-based approach to simplify the transformation definition process. Whether or not XML documents conform to a given document type definition (DTD) or XML Schema, typing information can be inferred and represented by so-called type graphs. Any XML document can therefore be represented as a typed graph and transformed in a rule-based manner, as in XSLT. The differences toward XSLT are mainly that rules can be depicted visually, are more abstract (so the order of execution does not depend on target document), IDREFs are dealt with much more naturally, and because of typing, the transformation output is guaranteed valid with respect to the target schema.

Often transformations between XML formats are needed back and forth, e.g. a UML model is translated to some semantic domain (for example Petri nets) to do some validation and the result which might be a change proposal, has to be translated back. We show that graph rules can be automatically reversed in certain cases, to formulate a reverse XML transformation.

The new approach for XML transformations has been tested at a variety of different transformations. Throughout this paper we discuss the transformation of class diagrams in XMI [18] format to entity-relationship diagrams in WebML [13] format, and back.

The paper is organized as follows: Section 2 introduces to the main concepts of XML and XSLT and illustrates them at the running example, an XML transformation from XMI to WebML. Section 3 gives an introduction into the basic graph transformation concepts which is used in section 4 to define our graph-based approach to XML transformation. This approach is applied to the running example in section 5. Thereafter, we discuss the possibilities to reverse XML transformations automatically in section 6. Related approaches and a short conclusion can be found in section 7.

2 XML and XSLT

XML Documents. The Extensible Markup Language (XML) [14] is a simple, very flexible text format derived from SGML (ISO 8879 [11]). Originally designed to meet the challenges of large-scale electronic publishing, XML is playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. XML documents are composed of markup and content, a snippet of an XML document is shown below. This example is an extract of a WebML

(Web Modeling Language [13]) document representing an Entity-Relationship diagram.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE WebML SYSTEM "WebML.dtd">
<WebML xmlns:auxiliary="http://www.webml.org/auxiliary"
       xmlns:graphmetadata="http://www.webml.org/graphmetadata"
       xmlns:presentation="http://www.webml.org/presentation"
       siteName="Acme" version="3.0.18">
  <Structure graphmetadata:go="Structure_go" id="Structure">
    <ENTITY auxiliary:testCaseCount="20"
        graphmetadata:go="User_go" id="User" name="User">
        <ATTRIBUTE id="userName" name="UserName" type="String"/>
        <ATTRIBUTE id="password" name="Password" type="Password"/>
        <ATTRIBUTE id="email" name="EMail" type="String"/>
        <RELATIONSHIP id="User2Group" inverse="Group2User" maxCard="N"
         minCard="1" name="User_Group" roleName="User2Group" to="Group"/>
         <RELATIONSHIP id="User2DefaultGroup" inverse="DefaultGroup2User"
         maxCard="1" minCard="1" name="User_DefaultGroup"
          roleName="User2DefaultGroup" to="Group"/>
    </ENTITY>
        ...
  </Structure>
</WebML>
```

The basic kinds of markup which can occur in the XML document content are the following:

- *Elements* are indicated by opening and closing tags (with angle brackets) and may contain other nested elements. If they don't they may also be written as a single in-line tag (e.g. <elem/>).
- *Attributes* are pairs composed of a name and a quoted-value inside start-tags after the element name.

Additionally, entities, comments, and CDATA sections are allowed as building blocks of XML (besides processing instructions).

XSL Transformations. Two W3C Recommendations, XSLT and XPath (the XML Path Language [15]), are provided to allow for transformation of a source XML document into another document written in any language. We use XSLT, which itself uses XPath, to specify how an implementation of an XSLT processor is to create our desired output from our given marked-up input. XML documents are represented as trees: XSLT provides constructs to navigate through nodes, iterate, and eventually produce new nodes in the output document, XPATH provides a way to select or express conditions regarding a node given a starting context of application. XSLT is a declarative language, the XSLT processor is not told how to perform the transformation, rather XSLT describes the expected result with respect to the source document structure. This allows a stylesheet to be applicable to a wide class of documents that have similar source tree structures.

There are two approaches to stylesheet design: 'push' and 'pull'. In the first one, the XSLT processor is instructed with templates (rules) to be performed when, during parser navigation, a certain element is encountered. It is called push because each node visited by the parser is "pushed" through the stylesheet to be caught by template rules. The output will be dictated by the source document. The push style is considered by many experts the most scalable approach, although some critics claim that code maintenance is hard. Push is the only way to go when the order in which XML elements will be encountered by the parser is not known a priori, like in text-oriented XML documents.

In the pull approach instead, source document nodes are selected ("pulled") from the source document by means of XPATH expressions as they are needed. The pull approach is usually composed by a single template containing a list of steps to perform, this more declarative approach is preferred by developers that never really got too much acquainted with functional programming style at the bottom of XSLT [9]. Pull is better suited for data-oriented documents as the developer can somehow anticipate the order of the information.

Most XSLT stylesheets use a combination of both approaches, the most common practice has push templates containing some pull instructions. In the code snippet below we use a template to transform a UML Association from XMI into a WebML Relationship. It uses a template to match a UML:Class (push) and produce an ENTITY element with the appropriate attributes. As in WebML a RELATIONSHIP element has to be nested inside an ENTITY, in this transformation we're forced to use the pull approach in order to retrieve the related association information before the production of the closing ENTITY tag. Thus, the structure of the target document limits our choice of coding approach.

Associations in XMI are represented by a quite verbose tree, two nodes called AssociationEnd identify the end points of the association by means of the attribute "participant". The attribute contains a reference to the identifier of another XML element. References to IDs are very common in XML: they are called IDREFs, and provide a way to express relations between elements that differs from nesting as it supports multiple cardinalities. The retrieval of all the association instances that end up in the UML:Class we are currently matching has to leverage the IDREF in attribute "participant" of element AssociationEnd. Therefore the apply-templates statement of line 4 uses an XPATH expression to select all association ends having an attribute called "participant" whose value is equal to the attribute "xmi.id" of the XML element we are currently matching. Note how the XPATH expression also considers the navigation path from the current element to the element we want to match. We could also have used a more general navigation path (worsening parser performance) or a "key" construct if we wanted to match all UML:AssociationEnd elements no matter their position in the source document. The example we provided is fairly simple, but gives a basic idea of the way IDREFs are handled in XSLT. Transformations that require navigating chains of IDREFs are much more complex and require either declaration of multiple keys, usage of variables, or invocation of multiple templates. Consider for instance the existence of the attribute "package" on the

UML:Class element being an IDREF to a UML:Package ID. If for any reason we wanted to translate into relationships only associations between classes in the same package we would necessarily have to use a key, a variable or a parametric template. In the following sections we will show the benefits of using graph transformation to handle IDREFs.

```
<xsl:template match="UML:Class">
   <ENTITY name="{@name}"id="{@xmi.id}">
     <xsl:apply-templates/>
     <xsl:apply-templates
       select="../*/*/UML:AssociationEnd
               [@participant = current()/@xmi.id]"/>
   </ENTITY>
</xsl:template>

<xsl:template match="UML:AssociationEnd">
  <RELATIONSHIP id="{@xmi.id}" name="{@name}" roleName="{@name}">
     <xsl:attribute name="inverse">
        <xsl:value-of select="../UML:AssociationEnd
             [@xmi.id != current()/@xmi.id]/@xmi.id"/>
     </xsl:attribute>
     <xsl:attribute name="maxCard">
        <xsl:value-of select="UML:AssociationEnd.multiplicity/
               UML:Multiplicity/UML:Multiplicity.range/
               UML:MultiplicityRange/@upper"/>
     </xsl:attribute>
     <xsl:attribute name="minCard">
           ...
     </xsl:attribute>
     <xsl:attribute name="to">
        <xsl:value-of select="../UML:AssociationEnd
               [@xmi.id != current()/@xmi.id]/@participant"/>
     </xsl:attribute>
  </RELATIONSHIP>
</xsl:template>
```

3 Graph Transformation

Graphs are a general means to represent any kind of data structures. Especially, they are well-suited to show the structure of XML documents. Visualizing an XML document by a graph, it usually resembles a DOM tree and can be enhanced by edges which represent references to other identities, in addition. For an example, see Fig. 1 where part of a WebML document is visualized.

If XML documents conform to a given DTD or XML Schema, this typing information can be represented by typed graphs. The DTD or XML Schema is translated to a type graph which looks similar to class diagrams (without additional constraints). As in object-oriented modelling, types can be structured

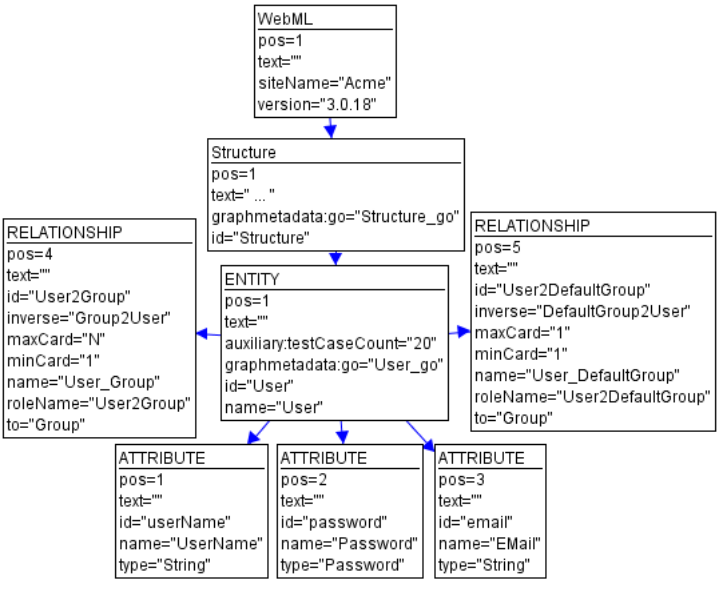


Fig. 1. Graph which represents the example WebML document in Section 2

by an inheritance relation [6]. Instances of a type graph are structure graphs equipped with a structure-compatible mapping to the type graph.

Formally, structure graphs are described by typed attributed graphs [7]. An attribute is declared just like a variable in a conventional programming language: we specify a *name* and a certain *type* for the attribute, and then we may assign any *value* of the specified type to it. All graph objects of the same type also share their attribute declarations, i.e. the list of attribute types and names; only the values of the attributes may be chosen individually. From a conceptual point of view, attribute declarations have to be considered as an integral part of the definition of a type. In theory [7], the attribute values are defined by separate data nodes which are elements of some algebra. In the AGG [1] tool, the attribution is based on Java (see below).

A *graph transformation rule* $r : L \rightarrow R$ consists of a pair of T-typed graphs L, R such that the union $L \cup R$ is defined. In this case, $L \cup R$ forms a graph again, i.e. the union is compatible with source, target and type settings. The left-hand side L represents the pre-conditions of the rule, while the right-hand side R describes the post-conditions. $L \cap R$ defines a graph part which has to exist to apply the rule, but which is not changed. $L \setminus (L \cap R)$ defines the part which shall be deleted, and $R \setminus (L \cap R)$ defines the part to be created. To make sure that newly created items are not already in the graph, we have to generate new vertex and edge identifiers whenever a rule is applied. Formally, for each application a new rule instance is created. Furthermore, a rule may specify attribute computations.

For this purpose, the rule graphs can be attributed by elements of term algebras which are instantiated by concrete values in the graphs when the rule is applied.

Two sample rules are given in Figures 3 and 5, created with AGG. Both figures show the LHS (left-hand side) L and RHS (right-hand side) R separately. All elements of $(L \cap R)$ are numbered correspondingly in L and R. Both rules do not delete anything, thus all elements in the LHS are numbered. The non-numbered elements in the RHS are the elements to be created. Both rules use a lot of variables as attribute values which indicates that arbitrary values are allowed. If several attributes have the same variable as value, the corresponding matched values in the host graph have to be equal. This is the case, e.g. in the rule in Figure 5 where attribute xmi.id of node 14:UML:Class has the same variable as value as node attribute participant in node 4:UML:AssociationEnd.

A *graph transformation step* is defined by first finding a match m of the left-hand side L in the current host graph G such that m is structure-preserving and type compatible. If a vertex embedded into the context, shall be deleted, dangling edges can occur. These are edges which would not have a source or target vertex after rule application. There are mainly two ways to handle this problem: either the rule is not applied at match m, or it is applied and all dangling edges are also deleted.

The applicability of a rule can be further restricted, if additional application conditions have to be satisfied. A special kind of application conditions are *negative application conditions* which are pre-conditions prohibiting certain graph parts.

Performing a graph transformation step with rule r at match m, all the vertices and edges which are matched by $L \setminus (L \cap R)$ are removed from G. The removed part is not a graph in general, but the remaining structure $D := G \setminus m(L \setminus (L \cap R))$ still has to be a legal graph, i.e., no edges should be left dangling. This means if dangling edges occur during a rule application, they have to be deleted in addition. In the second step of a graph transformation, graph D is glued with $R \setminus (L \cap R)$ to obtain the derived graph H. Since L and R can overlap in a common graph, its match occurs in the original graph G and is not deleted in the first step, i.e. it also occurs in the intermediate graph D. For gluing newly created vertices and edges into D, graph $L \cap R$ is used. It defines the gluing items at which R is inserted into D. A *graph transformation*, more precisely a graph transformation sequence, consists of zero or more graph transformation steps.

Given a host graph and a set of graph rules, two kinds of *non-determinism* can occur: first several rules might be applicable and one of them is chosen arbitrarily. Second, given a certain rule several matches might be possible and one of them has to be chosen. There are techniques to restrict both kinds of choices. Some kind of control flow on rules can be defined by applying them in a certain order or using explicit control constructs, priorities, etc. Moreover, the choice of matches can be restricted by specifying partial matches using input parameters. A common form of controlled rule application is the following one: One rule is selected from outside (e.g. the user) and triggers the application of

a number of other rules which become applicable after the first rule has been applied.

The graph transformation approach presented is supported by AGG [1] which is an integrated development tool for typed attributed graph transformation, implemented in Java. It offers the visual development of graph transformation systems including visual editing and simulation as well as a number of validation tools. The internal graph transformation engine can also be used by a Java API and thus, can be integrated into other tool environments. Several XML based input and output formats are available to the integration of AGG with other tools.

4 The Graph-Based Approach

The approach we propose aims at simplifying the process by letting the developer design the transformation visually and abstracting from document structure and element production order.

Relation between XML Documents and AGG Graphs. To be able to use graph transformation for the transformation of XML documents, there must be translations between XML documents and graphs. A simple solution is to provide universal XSL transformations from XML documents (without DTD or XMLSchema) to AGG graphs in the proprietary XML format for AGG, GGX, and back from GGX to XML. Once provided the user can completely concentrate on graph transformation and does not have to deal with XSL transformations at all. This idea can be extended to XML documents which conform to a DTD or XML Schema. In this case, the universal XSL transformation also transforms the DTD or XML Schema into a corresponding type graph. In this case the type graph may be enhanced by stronger constraints such as multiplicities.

These XSL transformations are applicable to any XML documents. A resulting AGG graph shows the structure of the corresponding XML document and resembles a DOM tree enhanced by additional edges which represent references to other identities.

The translation between XML documents and AGG graphs can also be obtained on the basis of a Java API for AGG which can be used to construct and read graphs.

XML Transformation by Graph Transformation. Describing an XML transformation by graph transformation, the source and target documents are visualized by graphs as discussed above. Performing XML transformation by graph transformation means to take the structure graph of an XML source document, and to transform it according to certain transformation rules. The result is the structure graph of the XML target document.

An XML transformation can be precisely defined by a graph transformation system $GTS = (T, R)$ consisting of a type graph T and a set of transformation rules R. The structure graphs of the source documents can be specified by a subset of instance graphs over a type graph T_S. Correspondingly, the structure

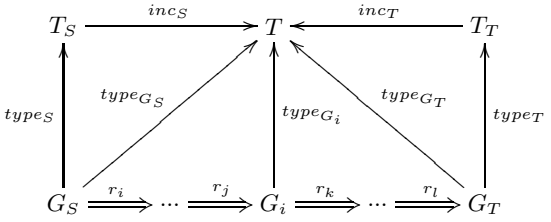


Fig. 2. Typing in the transformation process

graphs of the target documents are specified by a subset of instance graphs over a type graph T_T. Both type graphs T_S and T_T have to be subgraphs of the common type graph T. See Figure 2. Starting the XML transformation with instance graph G_S typed over T_S, it is also typed over T. During the transformation process, the intermediate graphs are typed over T. Please note that this type graph may contain not only T_S and T_T, but also additional types and relations which are needed for the transformation process only. The result graph G_T is automatically typed over T. If it is also typed over T_T, it fulfills the requirement to be valid.

5 Example: From XMI to WebML

In this section, we take up the running example again and show how graph transformation can be used to transform UML class diagrams in XMI format into entity-relationship diagrams in WebML.

The type graph for the transformation consists of three parts. Figure 4 shows the main section of type graph. The left part represents the type graph for WebML structures. The right part shows the type graph for XMI structures. In the middle, is one node type transf for relating XML nodes in both structures.

The transformation system contains five rules connecting nodes of the XMI document to newly created nodes in the WebML document (one rule for each element in the target document). Rules are quite simple and generally map a set of nodes (XMI is particularly verbose) into a target document node. Figure

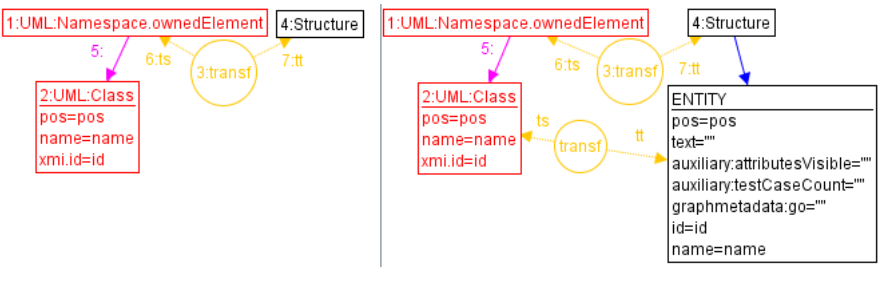


Fig. 3. Graph rule which translates classes to entities

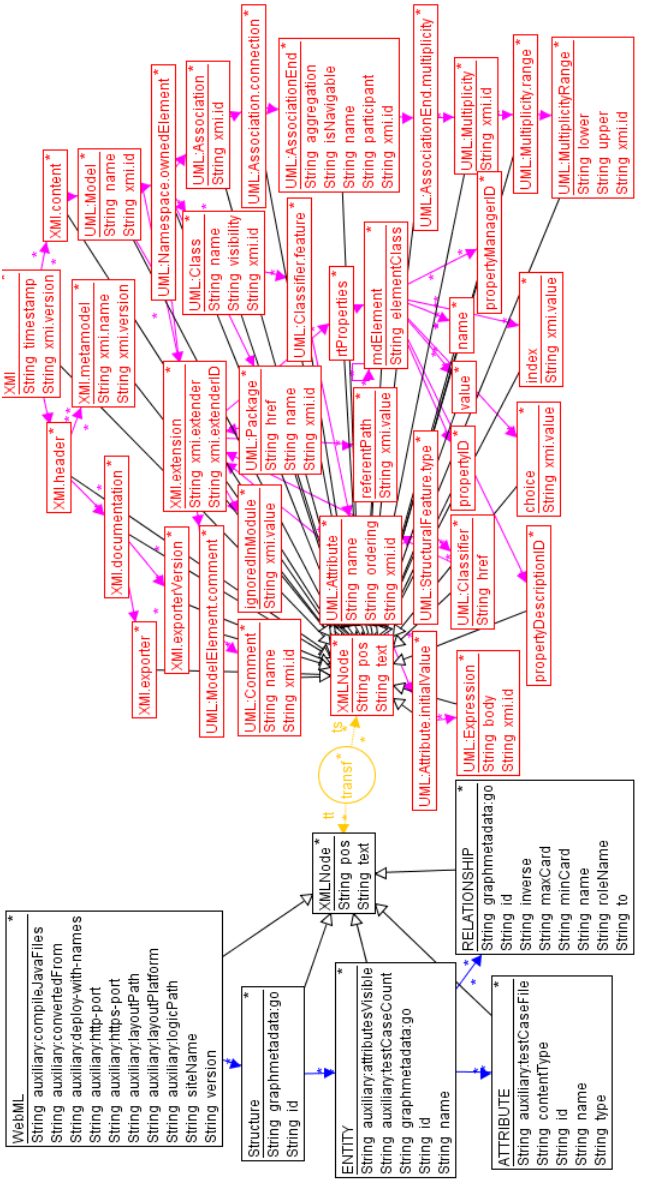


Fig. 4. Type graph of XMI to WebML transformation

3 shows the rule converting a class node into an entity node. The left hand side matches a UML:Class node child of a UML:Namespace.ownedElement, the latter has already been translated into a Structure element to which it is connected by a transf node and two edges. The RHS of the rule adds a new entity, connects it to the parent node with a child edge, and to the originating UML:Class node

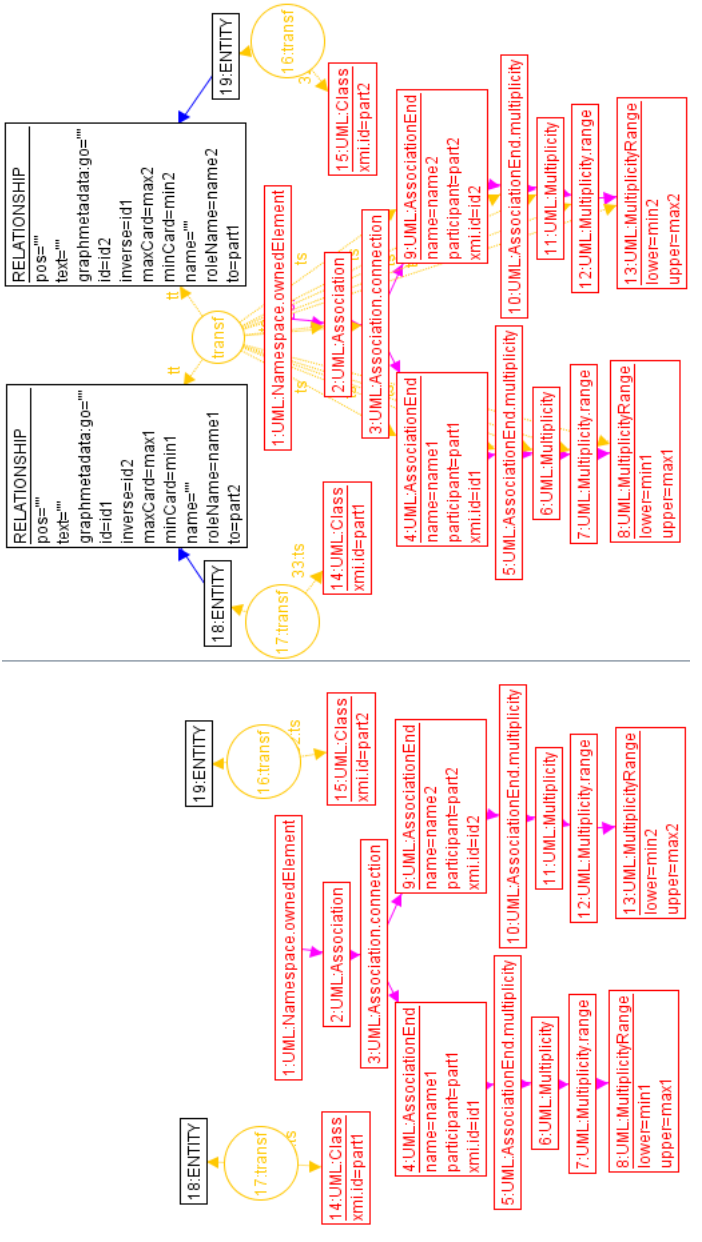


Fig. 5. Graph rule which translates associations to relations

via a transf node and edges. Since this transformation should be performed only once for each class, the rule is equipped with a negative application condition which is structurally equal to the RHS. That means before inserting a new entity for some class, we check that this class is not already related to some entity. This

visual approach simplifies the design of the transformation giving the user a clear representation of what each rule will produce.

The rule in Figure 5 is used to create two relationship nodes starting from a UML:Association subtree. In LHS, we search for a pattern consisting of an association with association ends which refer to the participating classes by attribute participant in UML:AssociationEnd nodes. The references are enforced by variables part1 and part2, to be matched with class IDs. Please note that this rule inserts two relationships, i.e. translates the association completely in one step, something not achievable in XSLT. Moreover, the use of variables to resolve IDREFs makes the rule clear at first sight. Again, this rule has a negative application condition structurally equal to the RHS, which prevents the rule from being applied twice to the same association.

In addition to the example presented here, we successfully experimented our approach also in transforming XML graph representation into Scalable Vector Graphics (SVG), rendering XML documents in HTML and reverse, performing WebML model to Struts configuration files transformation. We report on these experiments as examples for graph transformation applications on the AGG home page [1].

Discussion. The advantages of using graph rules instead of an XSLT transformation are multiple: first of all the result graph is typed, therefore enforcing the validity of the output with respect to the target document schema. This can be obtained in XSLT only by using schema-aware processors. Second, the representation of the type graph allows for an easier visual definition of the rules by matching subtrees, rule application conditions and behaviour are evident at first sight. Third, the use of variables (or edges) to deal with IDREFs is much more straightforward than any other construct in XSLT as we don't have to look for elements considering current context but we can naturally compose chains of IDREFs without having to declare multiple keys or complex (context-dependent) navigation XPATHs. The disadvantages of using graph transformations reside in the fact that in general the matching of the LHS of a rule in an instance graph is NP-complete, and basic graph transformation systems don't have a "natural" way of expressing a sequence of execution. But more elaborated forms of graph transformation systems provide different kinds of control on rule applications, as e.g. execution layers, priorities, control flows, etc. Some powerful constructs could also be inspired by XSLT (e.g. implicit and explicit rule priorities) or the new XSLT2 proposal [17], such as the "xsl:next-match" instruction.

6 Reversal of XML Transformations

Automatic Reversal of Graph-Based XML Transformations. Due to the fact that they are at a higher level of abstraction, graph-based XML transformations are composed of rules that do not depend on the parsing order of the source document or order of nesting of the output. For this reason, under certain conditions, they can be automatically reversed to produce the inverse transformation, that

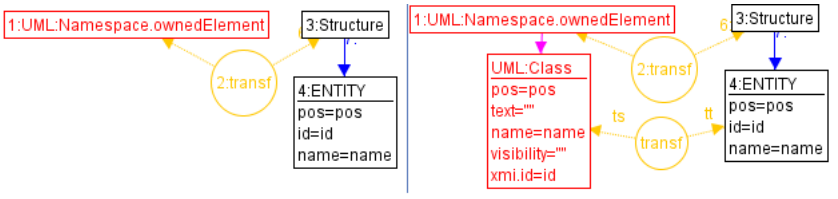


Fig. 6. Graph rule which translates entities to classes

is from the target document structure to the source one: this is not achievable with XSLT where each transformation is inherently uni-directional.

Obviously, to be fully reversible, a transformation would need to be information lossless: alas, this is not the case in most real applications where target documents simply do not require some data. Anyhow, the proposed approach is able to provide a reverse transformation as close as possible. Moreover, once a transformation is performed, the result graph preserves information about elements related by the transformation.

The graph rules performing the reverse transformation are based on the same type graph as the original rules, no changes are needed. The first observation when reversing rules is that all XML transformation rules we used are non-deleting: they only add elements. All transformation rules have a context which contains a relation between source and target elements already established. This context is preserved in forward and backward transformation rules. In addition, the LHS of the forward rule contains some source part, while the backward rule contains some target part. As RHS of the backward rule we take the RHS of the forward rule. It remains almost unaltered as it represents the completed relation between the source document and the target one.

The computation of attribute values is inverted accordingly, with slight differences: Each attribute of a new element in a RHS must be provided with an initial value. If an attribute values cannot be restored, a default value has to be used. If target attributes are computed by functions on source attributes, in the reverse RHS, source attributes are calculated by inverse functions on target attributes.

Example: From WebML to XMI. This example shows one of the rules automatically obtained by inverting our example rules given in Section 5. Figure 6 shows the rule transforming an entity into a class, being the inverse rule of the one in Figure 3. The RHS of the rule is obtained from the original RHS by defining attributes of source document elements in terms (or functions) of target document attributes. As not all Class attributes are preserved in the WebML representation, just those attribute values that can be retrieved are used, therefore some Class attributes are left empty. Attribute values which were left empty in the Entity element of the original rule are discarded. The LHS is derived from the new RHS by deleting the new "transf" node and all the source document elements that were connected to it (and to no other "transf" node).

7 Related Work and Conclusion

Even though XSLT is a popular and well supported XML transformation language, other approaches might be better suited to perform some kind of transformations (or might better suit personal tastes). In the current paper we proposed a graph based approach that simplifies the specification of XML document transformations with respect to XSLT by being visual and independent of node writing order, by providing a "natural" way to deal with IDREFs and by allowing for a unique specification for bi-directional transformation. This makes the proposed solution especially suitable for the OMG's MDA methodology. We developed a prototype implementation of the approach based on AGG.

Different proposals exist for using visual approaches to query, perform syntax-checking, infer DTDs and schemas, and transform XML documents. XML-GL [5] uses graphs both for representing XML documents and queries on them, but it does not perform document transformation between different vocabularies. XQBE (XQuery by example [4]) provides a visual language to specify queries on XML documents and translates it into XQuery or XSLT. VXT [10] is a visual methodology to specify uni-directional XML document transformation, while XMLTrans [19] is a Java based transformation language. Xing [8] is a visual language to query XML documents. In [2] a graph grammar for inferring the DTD of an XML document is proposed. In [20] and [21] the authors use a context-sensitive graph grammar for both defining the schema of an XML document and the rules to translate it into another vocabulary. Bezivin et.al. [3] propose a model transformation approach to obtain tool interoperability in the context of certain applications. This approach shows some similarities to ours in the sense that it is based on EMF models and uses a more abstract transformation approach which is QVT-like [12].

Apart from using a different formalism w.r.t. other approaches our proposal performs DTD inference when needed, XML document transformation between different vocabularies with advantages w.r.t. XSLT regarding typing, visual matching and IDREFs, plus it allows reverse transformations. Future work will deal with performance issues and focus specifically on formalizing the requirements for a transformation to be fully reversible.

References

1. AGG Homepage. http://tfs.cs.tu-berlin.de/agg.
2. L. Baresi, E. Quintarelli. Graph transformation to infer schemata from XML documents. In Proceedings of the 2005 ACM symposium on Applied computing, pages 642 - 646, ACM Press, 2005
3. J. Bezivin, H. Bruneliere, F. Jouault, I. Kurtev. Model Engineering Support for Tool Interoperability. In Proceedings of 4th Workshop in Software Model Engineering at 8th Int. Conf. on Model Driven Engineering Languages and Systems, 2005.
4. D. Braga, A. Campi, S. Ceri. XQBE (XQuery by Example): a visual interface to the standard XML query language. ACM Transaction On Database Systems TODS, June 2005

5. S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, L. Tanca. XML-GL: A Graphical Language for Querying and Reshaping XML Documents. In Proc. of the Int. World Wide Web Conference, Canada, 1999
6. R. Bardohl, H. Ehrig, J. de Lara, and G. Taentzer. Integrating Meta Modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In M. Wermelinger and T. Margaria, editors, *Proceedings of FASE 2004*, pages 214–228, 2004.
7. H. Ehrig, U. Prange, and G. Taentzer. Fundamental Theory for Typed Attributed Graph Transformation. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Proceedings of ICGT 2004*, volume 3256 of *LNCS*, pages 161–177. Springer, 2004.
8. M. Erwig. Xing: A Visual XML Query Language. In Journal of Visual Languages and Computing, 14(1):5– 45, 2003
9. D. Novatchev. The Functional Programming Language XSLT - A proof through examples. http://www.topxml.com/xsl/articles/fp/ Nov. 2001
10. E. Pietriga, J. Vion-Dury, V. Quint. VXT: a visual approach to XML transformations. In Proceedings of the 2001 ACM Symposium on Document engineering, pages 1–10, ACM Press, 2001
11. SGML, http://www.w3.org/MarkUp/SGML/
12. Query/View/Transformation. QVT-Merge Group, version 2.0 (2005-03-02), 2005. http://www.omg.org/cgi-bin/apps/doc?ad/05-03-02.pdf
13. WebML, http://www.webml.org/
14. XML, http://www.w3.org/XML/
15. XPath, http://www.w3.org/TR/xpath
16. XSLT, http://www.w3.org/TR/xslt
17. XSLT 2.0, http://www.w3.org/TR/xslt20/
18. XMI, http://www.omg.org/technology/documents/formal/xmi.htm
19. D. Walker, D. Petitpierre, S. Armstrong. XMLTrans: a Java-based XML transformation language for structured data. In Proceedings of the 18th conference on Computational linguistics - Volume 2 pages 1136 - 1140, 2000
20. K. Zhang, D. Zhang. XML Transformations Through Graph Grammars IEEE International Conference on Multimedia and Expo, 2001.
21. K. Zhang, D. Zhang, Y. Deng. A Visual Approach to XML Document Design and Transformation. IEEE Symposium on Human Centric Computing Languages and Environments, 2001.

OMake: Designing a Scalable Build Process*

Jason Hickey and Aleksey Nogin

Computer Science Department,
California Institute of Technology,
{jyh, nogin}@cs.caltech.edu

Abstract. Modern software codebases are frequently large, heterogeneous, and constantly evolving. The languages and tools for software construction, including code builds and configuration management, have not been well-studied. Developers are often faced with using 1) older tools (like make) that do not scale well, 2) custom build scripts that tend to be fragile, or 3) proprietary tools that are not portable.

In this paper, we study the build issue as a domain-specific programming problem. There are a number of challenges that are unique to the domain of build systems. We argue that a central goal is compositionality—that is, it should be possible to specify a software component in isolation and add it to a project with an assurance that the global specification will not be compromised. The next important goal is to cover the full range of complexity—from allowing very concise specifications for the most common cases to providing the flexibility to encompass projects with unusual needs. Dependency analysis, which is a prerequisite for incremental builds, must be automated in order to achieve compositionality an reliability; it also spans the full range of complexity.

We develop a language for describing software builds and configuration. We also develop an implementation (called OMake), that addresses all the above challenges efficiently and portably. It also provides a number of features that help streamline the edit/compile development cycle.

OMake is freely available under the GNU General Public License, and is actively being used in several large projects.

1 Introduction and Problem Definition

The general objective of a build system is to automate the construction of a software product from a set of inputs. For example, the product might be an application executable, where the inputs are the source files; in this case, the executable is usually constructed by compiling and linking the source files. The software product might also have several parts, for example it might be a web site that is to be constructed from a set of source scripts and document files. The process of generating the product from the inputs is called *building* the product; each run is called a *build*; and the tool used to manage the build is called a *build system.*

* An extended version of this paper is available as a California Institute of Technology Technical Report CaltechCSTR:2006.001.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 63–78, 2006.

1.1 Specification of the Build Process

In general, we will assume that both the inputs and the results of a build are represented as files. A complete build is usually composed of several steps, including actions like 1) compiling source files, 2) linking object files to construct libraries or executables, 3) generating documentation, 4) and packaging the results. In the interest of modularity, and also to allow incremental builds, we would like to specify a build in terms of steps, where each individual step involves executing a script or application, such as a compiler, to generate a set of output files from a set of input files. We call the output files *targets*; the input files are called *dependencies*. In some cases we also refer to named tasks as targets.

We assume that each step can be specified as a *build rule* with the following parts.

- a set of *targets* to be built,
- a set of *dependencies*,
- a set of files, called *side-effects*, that may be modified during execution of the rule; the targets are always side-effects of the rule,
- a function or script, called the *build commands*, that may be called to construct the targets from the dependencies. We say that a rule is *executed* when its build commands are executed.

For the purpose of incremental builds, smaller steps are often preferable.

We further classify the dependencies: *explicit* dependencies are part of the rule specification, and *implicit* dependencies are all other factors that may affect the outcome of a rule execution. For example, if a dependency file.c contains a line #include "file.h", then file.h is an implicit dependency of the rule if it is not already explicit. Strictly speaking, the compiler binary is also a dependency.

A *build specification* for a project defines a set of build rules that form a *dependency graph*. The leaves of the graph are the files that do not appear as targets; they correspond to source files. A target is considered *up-to-date* if all of the following hold:

1. it has been built at least once, and the most recent rule execution was successful,
2. all of its dependencies are up-to-date,
3. the dependencies and commands have not changed since the previous time it was built,
4. none of the effects (the side-effects and the target itself) have changed since the previous time the rule was executed.

Leaf files are always up-to-date, if they exist. The task of a build system is to bring the desired targets up-to-date by executing a (preferably minimal) set of rules.

This definition of "up-of-date" has several noteworthy properties. First, it does not refer to such unreliable properties as file timestamps; even a file with a very recent timestamp may be considered out-of-date if it was produced by something external to the build system and the build system has no way of

knowing for sure whether it was produced correctly. Second, the definition explicitly states that the target has to be rebuilt when the corresponding command changes. This means that when a user updates the build configuration (for example, the compiler flags), the build system will be required to rebuild all the targets that have to be built differently under the new configuration (again, regardless of how fresh the timestamps are). Finally, it allows for not propagating the changes that do not affect the outcome. For example, if a program myprog depends on file.o, which in turn depends on file.c, then insignificant changes to file.c (such as a white space or comment change) will cause only file.o to be rebuilt, but myprog does not have to be rebuilt unless file.o changes too.

1.2 Constraints and Requirements

Not every build graph defines a well-formed project. We impose the following constraints.

1. The dependency graph must be acyclic.
2. Each target is the target of exactly one rule.
3. If the transitive dependencies of a rule are up-to-date, then executing a rule successfully brings the targets of the rule up-to-date.
4. Rules may be executed in parallel if their side-effects do not intersect.

The acyclic requirement is used to help ensure termination of the build. If termination is not a concern, the acyclic requirement can be relaxed, and the build process becomes a fixpoint calculation. The second and third requirements are constraints on the programmer of the build. In order for the build specification to be robust and maintainable, there must be exactly one way to build each target, and the command to build it must be correct. The final requirement is for performance and is not strict. If interferences between rules are not specified accurately, the build user is limited to serial rule execution.

2 Design Requirements

As specified, the build system implementation might appear to be a straightforward task of providing a solver that takes a dependency graph and executes rules in some order to bring all targets up-to-date. Indeed, the solver is reasonably straightforward and algorithmically unsurprising. The interesting issues are on either side of it. First, how should the dependency graph be specified; and second, how may build commands be specified and executed portably? Before answering these questions, we introduce the design requirements.

- There should be a single build specification (perhaps in multiple files) for an entire project.
- Build specifications must be configurable. In other words, it should be possible to parameterize them by properties like project requirements and versions, by the availability of tools, by the target platform, and other properties of the build environment.

- The build specification should be stable relative to project evolution. That is, maintenance of the build specification should be insignificant relative to the project development and maintenance effort.
- Specifications must be compositional. That is, the sub-specifications for the components of a project may be expressed independently, and combined without interference.
- The build system should be general, not specific to a particular application domain.
- The system should not require that all dependencies be stated explicitly; instead it should provide an automated mechanism for discovery of implicit dependencies.

These requirements rule out some naive solutions. For example, requiring that the programmer provide the full, literal, dependency graph is not possible because for large projects the specification would be large, repetitive, and difficult to maintain; in addition the specification would not be parameterizable or configurable.

What is clear is that the language of the build should be general enough to support specification definitions that are both concise and configurable. One approach is to use a general-purpose scripting language to construct the dependency graph. This is the approach taken, for example, in both Cons [10] (which uses Perl) and SCons [8] (which uses Python). However, the expressive power of these languages often acts to tempt programmers away from simple declarative specifications. In particular, there is no guarantee of compositionality in these languages; the build specifications in different parts of a project may exhibit unforeseen interference unless programmers are strictly disciplined.

We take the opposite approach, working from the bottom up, including features in the language only when they satisfy the design requirements. In the next section, we begin the task with rule specifications, and work towards each of the design goals.

3 Language

3.1 Rule Specifications

The primary goal of a build specification is to define a dependency graph, which is a set of build rules. A build rule has a set of targets, dependencies, side-effects, and some build commands. For this purpose, the rule syntax used in the ubiquitous Unix make program[1] is ideal. A rule has the following form, where the targets, dependencies, and side-effects are lists of filenames, and the commands define a script to build the targets from the dependencies (we will use standard "command-line" syntax for the commands). The notation $[\cdots]$ indicates that the syntactic form is optional—the brackets are not part of the concrete syntax.

```
targets: dependencies [ :effects: side-effects ]
    commands
```

[1] Although indentation is not restricted to tabs.

For example, using a standard shell syntax for the commands, the following rule specifies how to construct a grammar implementation from its specification using the yacc application.

```
grammar.h grammar.c: grammar.y :effects: y.tab.c y.tab.h
    yacc grammar.y
    mv y.tab.c grammar.c
    mv y.tab.h grammar.h
```

We call these *explicit rules* because the targets, dependencies, and side-effects are all specified with explicit filenames.

Implicit rules. One of the main issues with explicit rules is that they are overly verbose and repetitive. For example, a project might have many C source files that are all the be compiled with the cc compiler, and it is inefficient to define a separate rule for each file. *Implicit rules* address the issue by defining *rule patterns*. In an implicit rule, the % character represents a "wildcard" pattern that stands for an arbitrary string of text. All occurrences of the wildcard represent the same string throughout a rule (in other words, the wildcard is universally quantified).

With implicit rules, generic rules can be specified by pattern matching. For the example in the previous paragraph, the following implicit rule specifies that the cc compiler may be executed to compile any file with suffix .c, producing a file with a .o suffix.

```
# Use cc to compile a .c file, producing a .o file
%.o: %.c
    cc -c -o %.o %.c
```

Rule selection. One of the requirements of the dependency graph is that there be exactly one rule for each target. For explicit rules, it is an error for a file to occur as the target of more than one explicit rule. However, with implicit rules, it is desirable to allow multiple potential matches (although at most one rule may be selected for use in the dependency graph). For example, one might define an implicit rule that specifies a "default" build action, and then define explicit rules for any special cases where the default is inadequate.

We define rule selection by policy. Given a specific target with name T,

- if T is the target of an explicit rule, that rule is used,
- otherwise, if T matches an implicit rule in scope (we define the concept of scope in the next section), then the most recently defined implicit rule that matches T is chosen,
- otherwise, the source file T must exist, and it is a leaf in the dependency graph.

3.2 Variables, Scoping, Compositionality, and Parameterization

Even with implicit rules, there is a great deal of duplication. Many related rules (such as compiling and linking rules) will want to ensure that they are constructed by the same application or compiler, with the same options. In addition,

it is usually desirable to allow the specification to be easily reconfigured. The obvious solution here is to introduce variables that represent values that may be used in multiple rules. Once again, we adopt the standard syntax, using the notation $\$(\cdots)$ for variable references, and single-line definitions using =. For example, the following two rules specify that program p is generated by compiling and linking two files x.c and y.c. The compiler is defined with the variable CC, and the options are defined with the CFLAGS variable.

```
CC = gcc
CFLAGS = -g
%.o: %.c
    $(CC) $(CFLAGS) -c -o %.o %.c
p: x.o y.o
    $(CC) $(CFLAGS) -o $@ $+
```

For convenience, within the rule commands, the variable $@ is defined as the target of the rule, and $+ are its explicit dependencies.

Scoping and compositionality. Before the introduction of variables, build specifications were purely declarative and compositional. That is, suppose two developers had defined build specifications for two sub-projects. We could combine their build specifications simply by concatenating them. As long as the two did not share targets (which might violate target uniqueness), the combined specification would be valid and correct relative to the sub-project specifications.

With variables, the situation changes. Suppose the concatenated specification happens to share variables, as follows.

```
# Developer 1
CC = cc
CFLAGS = -g
file1.o: file1.c
    $(CC) $(CFLAGS) -c file1.c
```

```
# Developer 2
CC = gcc
CFLAGS = -O6 # (unsafe in general)
file2.o: file2.c
    $(CC) $(CFLAGS) -c file2.c
```

In the concatenated specification, the CC and CFLAGS variables are defined twice—which value is the right one? Furthermore, it is known that the -O6 option is a bit dangerous with gcc. If the values defined by developer 2 "win," then the code for developer 1 might be compromised.

So far, we have more-or-less adhered to tradition. The GNU version of make [9] includes explicit rules, implicit rules, and variables in the form we have described. However, at this point we take a radical departure. In make/GNUmake, one of the values would "win," and for example, CFLAGS would be either -g (debug mode) or -O6 (unsafe-optimizing mode) for the entire project, with unintended consequences for the other developer.

Our approach is radical to some and natural to others. As we see it, both developers are right, and the correct interpretation is the pure one ("purity" in the sense of functional programming). That is, the definition of a variable, like CFLAGS = -O6, is a *definition*, not an *assignment*. Each rule is a *closure* (another

concept from functional programming) that pairs the rule with its environment. The file file1.c is compiled with cc -g, and the file file2.c is compiled with gcc -O6.

The choice of pure specifications has two major consequences. As a benefit, specifications are always compositional, because there is no way for one part of a project to interfere with another by side-effect. In consequence, there is no easy way in general for a programmer to collect global information through side-effects on a global variable. As an aside, the pure *vs.* impure debate has been present for many decades, but we argue that for build specifications in particular, impure programming is more often by accident than by intention, and the benefits of compositionality far outweigh the benefits of shared, mutable state.

3.3 Programming and Configurability

At this point, we have explicit rules, implicit rules, and variables. While this might encompass many applications, it is still insufficient. For example, while implicit rules can be used to describe a great deal of build procedures, they cannot describe rules in which the dependency names are not literally textually related to the targets. In addition, build specifications are not easily configurable, because there is no way to state that the set of build rules depends on compile-time configuration parameters.

To address these issues, we introduce a simple core programming language with functions, function application, and conditionals.[2] The syntax of our language is shown in Figure 1. It is still modeled on the language for GNU make, with user-defined functions. Here, we use braces $\{p\}$ to represent block structure as defined by indentation—that is, the program p must be indented from the enclosing context; the braces do not appear in the concrete syntax.

The structure of the language is quite simple. A program is a sequence of statements, and each statement is either a command line to be executed by the shell, or a language directive such as a variable/function definition, function application, conditional expression, or rule definition. A rule definition may include options, like the :effects: we have seen earlier. There are other options as well, including :value: for dependencies on computed values, rather than files.

Functions, simplification, and configuration. The use of functions can often significantly simplify build specifications. As an example, let's consider the problem of building a static library from a set of object files. The rules to do this differ slightly between Win32 and Unix platforms, so we would like to define a function that computes the appropriate build rule based on the platform. Consider the following program.

[2] Expressivity is a double-edged sword. The traditional make/GNUmake programs do not include user-defined functions, most likely because such languages are Turing complete—even termination is not decidable. However, the loss of completeness in these languages has a heavy cost, leading many programmers to resort to meta-programming, such as imake [2], autoconf/automake [6, 7], or other build specification generators.

e ::=		expressions
	text	text
|	$\$(v)$	variables
|	$\$(v\ e_1, \ldots, e_n)$	function application
|	$e_1 e_2$	concatenation
s ::=		statements
|	e	shell commands
|	$v = \text{e}$	variable definitions
|	$v(v_1, \ldots, v_n) = \{p\}$	function definitions
|	$v(e_1, \ldots, e_n)$	function applications
|	$\textbf{if} e\{p_1\}\,\textbf{else}\{p_2\}$	conditionals
|	$\textbf{section}\{p\} \mid \textbf{export}$	scoping directives
|	$e_{targs} : e_{deps}\,(\textit{:option:}\ e_{opt})^*\ \{p\}$	rule definitions
p ::=		programs
|	ϵ	empty program
|	$\begin{matrix} p \\ s \end{matrix}$	sequencing (line endings act as sequence separators)

Fig. 1. The build programming language

```
# Platform-independent library construction
StaticLibrary(target, deps) =
    if $(equal $(OSTYPE), Win32)
        ofiles = $(addsuffix .obj, $(deps))
        $(target).lib: $(ofiles)
            lib /Fo$(target).lib $(ofiles)
    else
        ofiles = $(addsuffix .o, $(deps))
        $(target).a: $(ofiles)
            rm -f $(target).a
            ar cq $(target).a $(ofiles)
# An example library with 3 object files
StaticLibrary(mylib, file1 file2 file3)
```

The StaticLibrary function takes two arguments. The target is the name of the static library, and the deps are the object files to be included. Both arguments are provided without suffixes, since the actual file suffixes depend on the platform. The first step in the function is to determine the platform using a conditional. The OSTYPE variable defines the name of the platform, and the builtin equal function is used to determine if the platform is Win32. If so, the library has the .lib suffix, the object files have the .obj suffix, and the application for constructing the library is called lib. The addsuffix function is used to append the suffix to each of the names in the deps argument. The other case is similar.

The construction of a static library is now reduced to a single function call that specifies only the name of the library and its dependencies, with the usual benefits. The platform-dependent configuration is now located within a single

function, and the remainder of the build specification can be significantly simplified and need not be cluttered with platform tests. By defining a general set of such functions in a shared standard library, we obtain very concise specifications for simple projects.

Scoping and block structure. The introduction of block structure imposes a new twist on scoping. For example, in the previous section the StaticLibrary function defined the ofiles variable (twice). According to our scoping policy, these two definitions do not conflict since rules always use the most recent variable definitions in scope. The next question is whether the ofiles variable remains defined after the StaticLibrary function is called. Clearly, doing so would be undesirable because it would violate the abstraction provided by the function.

We adopt the usual scoping policy where each block in the program defines a scope, scopes are nested, and variables defined in inner scopes are not visible to outer scopes. Syntactically, blocks are determined by indentation, so for our example, the ofiles variable is *not* defined after the StaticLibrary function is called, because it is defined within an inner scope.

The **section** allows the introduction of a new nested block (with its corresponding scope), and it is frequently used to isolate variable definitions that are valid in only part of a project. For example, the following code fragment illustrates the common usage. The syntax CFLAGS += -g is equivalent to the expression CFLAGS = $(CFLAGS) -g, so the inner value of CFLAGS is "-O -g".[3]

```
CFLAGS = -O # Pass the "optimizing" flag to the C compiler
...
section
    CFLAGS += -g # Also add the "debugging" flag for the following targets
    ...rules...
# CFLAGS has the original value "-O"
...rules...
```

When combined, purity and strict scoping can be awkward. For example, consider the following program fragment, where the intent is to define the name of the C compiler and its default options on a platform-dependent basis.

```
# The compiler and flags are platform-dependent
if $(equal $(OSTYPE), Win32)
    CC = cl
    CFLAGS = /DWIN32
    OSUFFIX = .obj
else
    CC = gcc
    ...
```

[3] In our implementation, the system state including environment variables and the current directory are handled similarly—the extent of modifications is limited to the current scope.

Unfortunately, this program does not work as expected because the variable definitions are not visible outside the conditional. The **export** directive is designed to export variable definitions from an inner scope to its enclosing scope, canceling the nested scoping status of the block. In the example, the problem is solved by placing an **export** as the final statement in the branches of the conditional.

```
if $(equal $(OSTYPE), Win32)
    CC = cl
    CFLAGS = /DWIN32
    OSUFFIX = .obj
    export
else
...
```

3.4 Functions and Dynamic Scoping

One configuration pattern that arises often is to define the parameters of a project as variables, using the variables to define the rules that describe how to build the project. For an example, let's consider the rules for building applications in Objective Caml [5], which have the following general form.

```
OCAMLC = ocamlc # Byte-code compiler
OCAMLCFLAGS = # Compiler options (initially empty)
# Compile an OCaml file
%.cmo: %.ml
    $(OCAMLC) $(OCAMLCFLAGS) -c %.ml
# Link a program
OCamlProgram(target, deps) =
    cmofiles = $(addsuffix .cmo, $(deps))
    $(target): $(cmofiles)
        $(OCAMLC) $(OCAMLCFLAGS) -o $(target) $(cmofiles)
```

In this definition, we intend these rules to be the *default* rules. For example, even though the default compiler options OCAMLCFLAGS are empty, sub-projects should be able to redefine the variable if they require particular options. This presents a problem because, as we have stated, rules use the "most recent" definitions for variables, and these definitions are apparently fixed. Furthermore, the number of parameter variables can be quite large, and it would be unreasonable to require programmers to memorize them all.

The solution here is to use a definition of "most recent" as the most recent *dynamic* definition, not the most recent static one. That is, we adopt the use of dynamic scoping, rather than static scoping. With dynamic scoping, users of a build library need only be aware of the variables that need to be specialized. For example, the following code-fragment illustrates the temporary redefinition of the OCAMLFLAGS variable. In this case, the OCamlProgram is called in a context where the "-g" options has been added to OCAMLCFLAGS.

```
...
section
    # Compile the program with "-g" flag
    OCAMLCFLAGS += -g
    OCamlProgram(myprog, file1 file2 file3)
```

3.5 Automated Dependency Analysis

One of our design objectives is that it should be possible to automate the inference of implicit dependencies. There are several reasons, but the most important is that rule re-use becomes much more difficult when every rule is required to state all of its dependencies explicitly.

Implicit dependencies arise through many factors, including references to files in source code or applications, and they may change frequently as a project is developed. Traditionally, programmers have used ad-hoc meta-programming techniques, using a tool/compiler to generate the set of implicit dependencies, and then grafting them into the build specification textually.

In fact, the tools for dependency analysis already exist in many cases, and it takes very little for the build system to support them. To illustrate, the following program fragment specifies a rule for dependency analysis of C program files. The .SCANNER target is a directive, in this case indicating that the implicit dependencies of an object file can be extracted by compiling the C source file with the -MM option.

```
.SCANNER: scan-%.c: %.c
    $(CC) $(CFLAGS) -MM %.c
%.o: %.c :scanner: scan-%.c
    $(CC) $(CFLAGS) -c -o %.o %.c
```

The target of a scanner rule, in this case scan-%.c, is called a *scanner-target*, and represents the dependencies generated by the build commands. The command itself prints the dependencies (for one or more targets) to its standard output in make format.

Given a normal rule with targets targets and scanner dependencies deps, the complete set of dependencies is the union of the explicit dependencies, the dependencies generated by the scanner rules for each individual dependency, as well as the scanner targets themselves, or

$$\textit{explicit-dependencies} \quad \cup \quad \text{deps} \quad \cup \quad \bigcup_{d \in \text{deps}} \textit{scanned-dependencies}(\text{targets}, d).$$

3.6 Managing Subprojects

Our final design goal is that there must be a single build specification for an entire project. In most software projects, the software codebase is divided among several subdirectories, often, but not necessarily, along the lines of the software components. Similarly, it is impractical for the build specification to be placed

in a single file—instead it is more desirable to partition the specification along directory lines.

We adopt guidelines as follows. Each project must have a single directory, called the "root directory" (this is usually the root directory of the project). The root directory contains a file named OMakeroot that defines the build specification. Each build file may contain references to other directories of the project using a rule of the form .SUBDIRS: $dir_1, \ldots, dir_n$, where each subdirectory $dir_1, \ldots, dir_n$ defines its own build specification in a file named OMakefile.

Semantically, a .SUBDIRS directive acts as program inclusion in a nested scope. That is, variables and rule definitions are passed down to subdirectories, but definitions within a subdirectory are not propagated back to the parent. This prevents interference between the build specifications in separate subdirectories, and preserves compositionality. In addition, the ability to inherit values means that parent directories can define default behavior that can be specialized within the subdirectories.

3.7 Language Summary

At this point, it is worthwhile to revisit the design goals to see whether we have achieved our objectives. One of our primary objectives is compositionality, which we help ensure through the use of a pure language with well-defined scoping rules (even parts of the system state are treated purely). While it *is* possible for a programmer to achieve interference externally, for example through the filesystem, the risk of inadvertent interference is greatly reduced.

Another of our goals is configurability and generality. In this case, although the language is designed specifically for builds, it is general enough to cover a wide range of tasks. The expressivity and simplicity of the language also help in maintaining the build system. We have developed several large projects using a variety of build systems including GNU make, Cons, and SCons. In our experience, specifications based on the designs presented here are significantly more concise and easier to maintain. In addition, the .SUBDIRS approach to linking subprojects (also a feature of the Cons and SCons systems), has been enormously helpful for constructing simple, maintainable build specifications.

Finally, automated inference of implicit dependencies is important for ensuring consistency and accuracy of builds. Our approach allows the leveraging of existing dependency analyzers. In addition, the fact that the .SCANNER rules have the same properties as the normal build rules allows the use of the full build specification machinery, leading to concise, simple, yet flexible, and powerful dependency analysis.

4 Implementation: The OMake Build System

We have implemented a build tool, called OMake, that follows the design requirements stated in the previous section. OMake is freely available at the OMake home page `http://omake.metaprl.org/` under the GNU General Public License, and is actively being used in several large projects.

The OMake system has four major components. The first of them is a compiler that translates OMake specifications from the source language to an *intermediate representation* (IR). The second part is the interpreter capable of evaluating OMake programs in their intermediate representation form. The third part of the system is the build manager that keeps track of targets, dependencies and build rules (in their explicit and implicit forms). The build manager is also responsible for instantiating the implicit rules when needed and scheduling the build commands for execution. Finally, the fourth component of the build system is the shell interpreter that is responsible for executing individual build commands, spawning external processes as necessary, and passing the control back to the IR interpreter for commands that are to be executed internally.

Compiler. When performing a build, the OMake systems begins by reading the specification files, starting with the OMakeroot file. Each file that is part of the project specification is parsed and the code is transformed into an *intermediate representation* (IR). The translation process is straightforward; the IR is a slightly simplified, slightly more explicit version of the source language; there is little difference between the two. For every specification file, the resulting IR is cached on disk; this allows skipping the parsing and compilation of that file on subsequent executions of OMake (provided the given file does not change, of course).

Interpreter. Once a specification file is read and its IR is generated (or loaded from the cache), the OMake interpreter *evaluates* the IR. For the most part, the interpreter implementation is fairly straightforward. One of the least trivial parts of the interpreter is its handling of the variable environments. The variable environment data structure is implemented as a functional immutable lookup table where updates operate by partially copying the table. Each time a rule (whether implicit or explicit) is encountered during evaluation, the interpreter passes the rule and the current variable environment to the build manager as a closure. The rule itself is not evaluated immediately. Thus, many versions of the environment will be saved by the build manager. This approach is made cost-effective through the use of functional data structures and extensive sharing.

Build manager. Once the build specification has been evaluated, control is passed to the build manager, which now has a complete collection of build rules. This is not yet a dependency graph because some of the rules are implicit, and automated dependency analysis has not yet been performed. The build manager is responsible for building the dependency tree and making sure that the goal targets[4] are brought up-to-date. Note that it is not always possible to fully discover the dependency tree before the build process starts—it may be the case that in order to discover the full set of implicit dependencies the build manager will need to execute a number of .SCANNER rules and those rules may in turn depend on targets that need to be built first. Because of this, the build manager constructs the dependency tree in parallel with the main build process.

[4] If the goal targets are not specified on the command line, the .DEFAULT target is the goal.

The build manager works by keeping a *worklist* of targets that need to be brought up-to-date, each marked with a *state* specifying how far along the build process for the particular target has progressed.

For each project, the build manager maintains a database where it stores information about each rule that was successfully executed, including the full set of dependencies, the commands text, side-effects, and targets. On successive runs, the database is used to determine which targets are already up-to-date.

The shell interpreter. For portability, and a degree of efficiency, OMake includes a built-in command interpreter (this *shell* can be used both as part of the build system and also as a standalone command interpreter). For the most part, this is a straightforward task, involving standard methods for process creation and management. However, one of our primary goals is to provide transparent portability—OMake should behave similarly on all platforms, and Win32 in particular is problematic. Among the difficulties are the lack of a fork system call, signals, process control, and terminal management. As a result, we developed a compatibility library that emulates most of these features. The use of functional, immutable data structures allows us to emulate the fork system call using threads without the need for address space duplication.

Built-in functions and standard library. OMake provides a broad set of functions for string and string arrays manipulation, input/output, including functions that mirror the Unix standard IO library.

In addition, OMake provides a set of higher-level functions that can be used in order to make a project work correctly on platforms like Win32 that do not provide Unix-style file processing tools. The toolset includes functions that mirror the core functionality of the Unix programs grep, sed, awk, and test.

As mentioned in Section 3.3, OMake includes a shared standard library of variables, functions and implicit rules that can be used to significantly simplify the build specifications for commonly used languages. Using the standard library, simple projects in languages like C, OCaml, and LaTeX can often be specified with just a few lines of code, sometimes as little as one line.

5 Related Work

On Unix systems, the make program [3], originally designed by Feldman in 1979, is the ubiquitous build system, especially for open source projects. There are at least two reasons why make retains its popularity. First, the model is extremely simple. Second, make does not require any particular project style, nor is it tied to any particular programming language.

Since 1979, software projects have grown tremendously in size, and new versions of make have been developed, notably GNU make [9]. The usual model with large projects is to split a project into multiple subdirectories, each with its own Makefile describing how to build files in that subdirectory. Although this is adequate in many cases, there are several issues with regard to scalability. First, dependency analysis is based on timestamps. When a file is modified in any way,

it may cause large portions of the project to be rebuilt even if the modification was innocuous (for example, a change to a comment). Second, dependency information is local to each Makefile in each subdirectory. One result of this is that the subdirectories must be built in a specific order, and the graph of dependencies between subdirectories must be acyclic. A third problem is that each Makefile may have to duplicate a substantial portion of code also used in other Makefiles (for example, one of the main features of the imake utility [2] is automated code duplication).

A number of tools address some of these limitations. The Jam build system [11] addresses the problem of cross-directory dependencies, by performing a global dependency analysis and automating it so that dependency information is always up-to-date. The Odin system [1] provides a truly global environment by using a single cache for each user. In addition, Odin provides extensive support for build variants based on the concept of a *derived object* that couples properties with a file name.

The Cons tool [10], written in Perl, the SCons tool [8], written in Python, go further by adding configurable dependency scanners and adopting the use of MD5 digests instead of file timestamps. In both Cons and SCons, the build system is closely integrated with the implementation language. That is, in order to use these tools effectively, one must write build specifications in Perl (for Cons), or Python (for SCons). One advantage is that the use of a general-purpose programming language can reduce the amount of code duplication. However, there are also disadvantages of these tools when compared with the make model. In make, there is a clear separation between the language of Makefiles and the implementation language (C). The make language was designed specifically for specifying builds—it is clear and concise, it is widely used, and it is easy to understand. The make language is better suited for build specifications than the Perl or Python languages.

The Ant build system [4] takes another approach, where the build specification is written declaratively as an XML specification. The Ant system allows extensions written in Java.

We believe that one of the principal features that distinguishes OMake from all of the above systems is the use of a language that is Turing complete, yet preserves compositionality of build specifications. In addition, the language preserves the basic spirit, model, and syntax of make, preserving its strengths while addressing limitations of scalability and reliability.

6 Future Directions

While we are very satisfied with the convenience provided by the OMake tool, there are a number of further enhancements that we are hoping to explore in the future.

One of them is support for fixpoint builds (where certain dependency cycles are allowed). Examples of projects that could benefit from this feature include building self-hosting compilers (when a new version of a compiler is built using

an older binary, one often wants to arrive at a fixpoint) and LaTeX compilation (one may need to re-run latex several times if the .aux file is changing).

In addition, we are investigating the use of modular namespaces as a means of further improving scalability.

While OMake has some initial support for distributing builds over several different computers, it will probably require additional work before it is fully usable.

Acknowledgements

The authors would like to thank the OMake user community at omake@metaprl.org for discussion and support for the OMake project. The authors would also like to thank the anonymous reviewers for their comments and suggestions.

References

1. Geoffrey M. Clemm. *The Odin System*. Morristown, New Jersey, 1994.
2. Paul Dubois. *Software Portability with Imake, Second Edition*. O'Reilly, 1996.
3. Stuart I. Feldman. Make-a program for maintaining computer programs. *Software - Practice and Experience*, 9(4):255–265, 1979.
4. Steve Holzner. *Ant: The Definitive Guide*. O'Reilly, 2nd edition, 2005.
5. Xavier Leroy. *The Objective Caml system release 1.07*. INRIA, France, May 1997.
6. David MacKenzie, Ben Elliston, and Akim Demaille. *Autoconf: Creating Automatic Configuration Scripts*. Free Software Foundation, November 2003. http://www.gnu.org/software/autoconf/manual/index.html.
7. David MacKenzie and Tom Tromey. *GNU Automake*. Free Software Foundation, September 2003. http://www.gnu.org/software/automake/manual/index.html.
8. Scons: A software construction tool. Home page http://www.scons.org/.
9. Richard M. Stallman, Roland McGrath, and Paul Smith. *GNU Make: A Program for Directing Recompilation*. Free Software Foundation, July 2002. http://www.gnu.org/software/make/manual/index.html.
10. Rajesh Vaidheeswarran. Cons: A Make replacement. Home page http://www.dsmit.com/cons/.
11. Laura Wingerd and Christipher Seiwald. Constructing a large product with Jam. In *ICSE '97: Proceedings of the SCM-7 Workshop on System Configuration Management*, pages 36–48, London, UK, 1997. Springer-Verlag.

Automatic Generation of Tutorial Systems from Development Specification

Hajime Iwata[1], Junko Shirogane[2], and Yoshiaki Fukazawa[1]

[1] Department of Information and Computer Science, Waseda University,
3-4-1 Okubo, Shinjuku-ku, Tokyo 169-8555, Japan
{hajime_i, fukazawa}@fuka.info.waseda.ac.jp
[2] Tokyo Woman's Christian University,
2-6-1 Zempukuji, Suginami-ku, Tokyo 167-8585, Japan
junko@lab.twcu.ac.jp

Abstract. Recent complicated software functions have made it difficult for end users to operate them. Thus, it becomes important to learn how to operate them easily and effectively. Employing a tutorial system is the most suitable approach for learning how to operate software functions. A tutorial system demonstrates the how to operate using the actual software. As a result, end users can learn the usage as if they were actually using the software. However, development of tutorial systems requires much time and costs. Therefore, we propose a method of generating tutorial systems based on use case diagrams, sequence diagrams and test cases. In our method, a generated tutorial system shows function names extracted from use case diagrams, the how to operate along with sequence diagrams, and text string input and item selection using data from test cases. The generated tutorial system is then added to the source code for use in AOP (aspect-oriented programming).

1 Introduction

Recently, computer usage has become widespread, and various tasks have been computerized. Many kinds of software have been developed with many functions. Therefore, software usage tends to be complicated due to these many functions. It becomes more difficult for end users to learn how to operate software. It is important to provide support methods by which end users can learn the operation of the software. Now, there are some support methods for end user learning, such as online manuals, help systems, animated demonstrations and tutorial systems. In this paper, among the many end user learning methods, we particularly focus on tutorial systems.

A tutorial system has been used by end users for learning how to operate software. A tutorial system demonstrates the usage on the running software. When end users learn the usage of the software using a tutorial system, they can learn the sequences of operations as if they were actually using the software without interruption.

There are some advantages of a tutorial system over other many support methods:

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 79–92, 2006.

– End users can easily understand the purpose of each operation and the relationships among the operations.
– End users can interactively learn the software operation with the tutorial system. Thus, they can understand software operation using the tutorial system better than using an animated demonstration system.

However, currently, there are few built-in tutorial systems for software. Because the structures of software have now become highly complicated and have many functions, the development of tutorial systems places a heavy burden on software developers, and the cost is high.

In this paper, we propose a method for automatically generating a tutorial system based on use case diagrams, sequence diagrams [1] and test data provided by developers. These diagrams and data are made in the software development process. A generated tutorial system is woven into the software using AspectJ [2]. Using our method, it becomes easier to develop a tutorial system.

2 Tutorial System in Our Method

As an example, a tutorial system of address book software is shown in Figure 1. The right-hand window is a window of the tutorial system generated by our method. The left-hand window is a window of the address book software.

The software into which a generated tutorial system is woven is called target software in this paper.

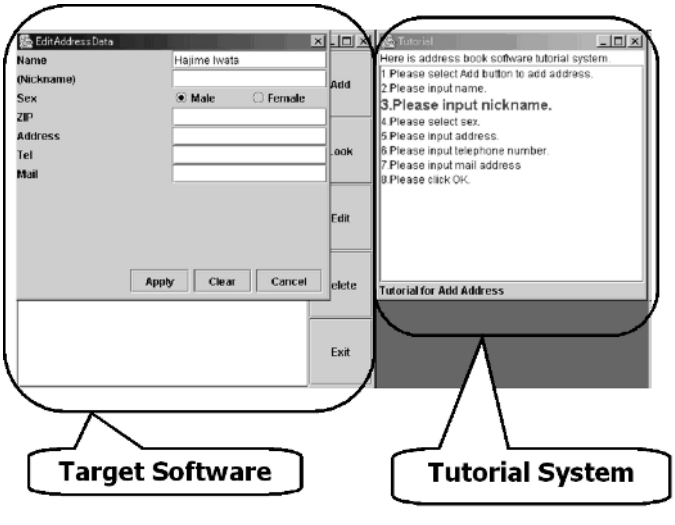


Fig. 1. Tutorial for address book software

In our method, the generated tutorial system, has the following features:

Showing Software Functions and Operations

In our method, the use case names are extracted from the use case diagrams for use as software function names in the tutorial system. The names of functions available in the software are shown in the tutorial system, and end users select the function they wish to learn. Then, the sequences of required operation steps extracted from sequence diagrams are shown step by step. The sequence of operation steps is described using itemized sentences in the tutorial system. End users can watch how the software function is operated, as well as the reaction of the software, continuously without any interruption. The sentence that explains the current operation is changed to have a vivid color and a large character size in the tutorial system window. Thus, it is easy to understand how to operate the software.

Showing Examples of Software Usage

The tutorial system shows a mouse pointer on a widget (Each part of a GUI, such as buttons, is called a widget.) in the target software at the same time highlighting the sentence explaining the current operation. At the same time, the required mouse operations are shown, such as clicking a button and choosing the radio button. End users can easily understand the sequences of operation steps required for the function by watching the software demonstration without interruption.

Demonstrating Input Texts

When it is necessary to input text for text fields, the tutorial system demonstrates an actual text input. The input text is not meaningless text but suitable

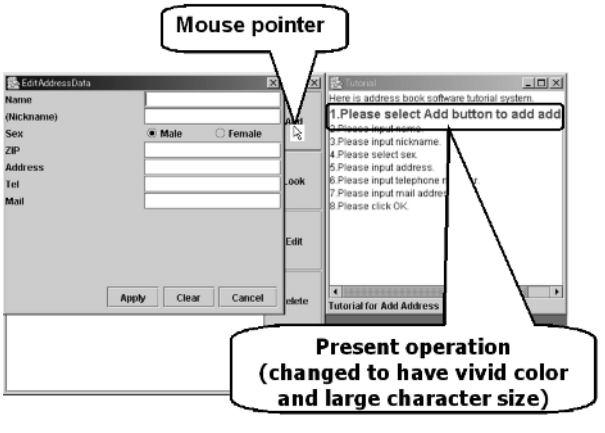


Fig. 2. Tutorial "Add Address"

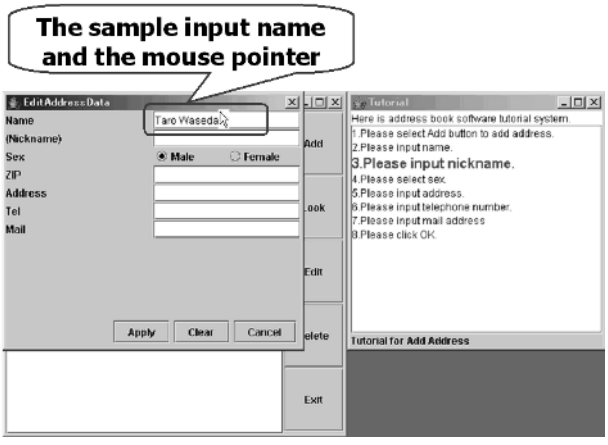


Fig. 3. Tutorial for sample input name

text extracted from the test data. We assume that the input text is the typical data in using the function. Therefore, end users can understand examples of actual data and how to input text for pertinent text fields.

In Figure 2, a user selects the tutorial "Add Address" is shown and how to operate for adding a new address.

The sequence of operations is displayed in the tutorial window as itemized sentences describing the steps in sequential order, and each step is numbered. When the end user selects the tutorial "Add Address", the explaining text which is "Please select Add button to add address." in Figure 2 is changed to have a vivid color and a large character size in the tutorial system window. Similarly, "Taro Waseda", the sample input name extracted from the test data, is entered automatically by the tutorial system. Also, the mouse pointer is moved to this text field. This example is shown in Figure 3.

3 Features of Our Work

3.1 Development Specification for Generated Tutorial System

In this paper, we propose a method for automatically generating a tutorial system for target software. Generating a tutorial system allows end users to efficiently learn how to operate the target software. The tutorial system is generated automatically on the basis of these specifications, which are use case diagrams, sequence diagrams and test data. Use case diagrams are used in the analysis phase of object-oriented software development, to describe software functions. Sequence diagrams are used for describing interaction between objects in order. An example of a use case diagram for the address book software in section 2 is shown in Figure 4, and an example of a sequence diagram in the case of the "Add Address" function is shown in Figure 5.

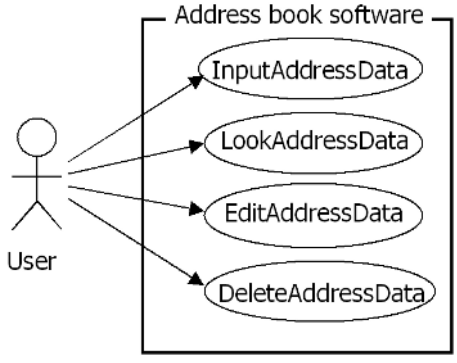


Fig. 4. Example of use case diagram

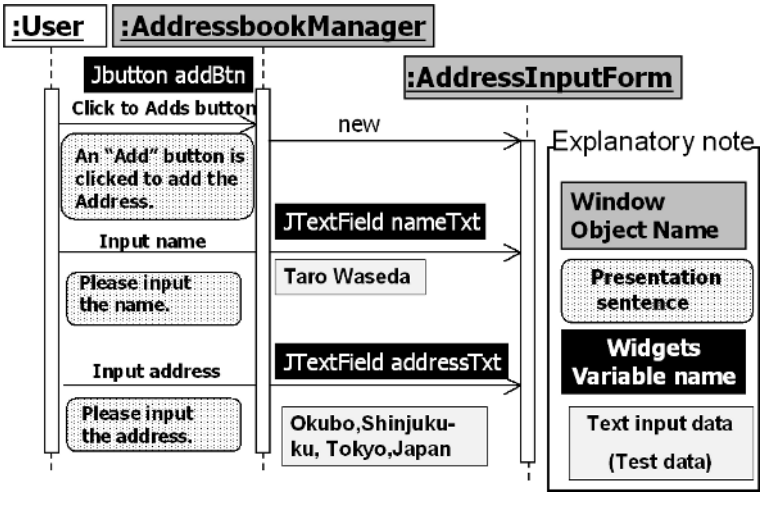


Fig. 5. Example of sequence diagram

In our system, diagrams described by the XMI (XML Metadata Interchange) [3] format are used. The XMI format is the standard file format for describing UML diagrams. Using the XMI format, use case diagrams and sequence diagrams provided by the existing UML modeling software can be generated for a tutorial system. Examples of UML modeling software supporting the XMI format are Rational Rose [4] and IIOSS [5].

In our research, the flows of procedures within use cases are described using sequence diagrams, and extra descriptions are added in the sequence diagrams. Target software developers combine the widget name, presentation sentence, and test data for input text, with the messages of sequence diagrams, and specify the window objects of sequence diagrams.

Window Objects

Target software developers are required to specify objects, which have widgets on GUI windows in the sequence diagrams. Thus, our system can specify the widgets that correspond to the objects, and our method can operate the tutorial system for the software. These objects are called window objects in our method. For example, ":AddressbookManager" and ":AddressInputForm" are the window objects seen in Figure 5.

Presentation Sentences

Target software developers are required to add extra descriptions to the messages of sequence diagrams. The extra descriptions are texts, which explain to users how to operate software functions. These extra descriptions are displayed to end users as explanations of the usage when the tutorial system is generated. For example, 'Click the "add" button to add the address' and "Please input the name" are the extra descriptions seen in Figure 5. These texts are called presentation sentences in our method.

Widget Variables

The names of the widget variables in a source code must be added to the messages of a sequence diagram. These widgets are operated by the end user, such as input text and select an item.

In our method, we assume the Java Swing set to be widgets. Typical widget variables that must be added are buttons (JButton), menus (JMenuItem), and check boxes (JCheckBox). For example, "addBtn", "nameTxt" and "addressTxt" are shown in Figure 5. However, widgets that are not operated by the end user, such as labels (JLabel) and panels (JPanel), do not need to be added to messages. Using a widget name, the coordinates of each widget in a window can be calculated. A demonstration of mouse pointer movement based on the coordinates of each widget is performed.

Also, each widget has a method that substitutes for the end user operation. For example, "JButton" has a method called "doClick()" which substitutes for a mouse click of the end user, and "JTextFields" has a method called "setText()" which substitutes for a data input of the end user. The tutorial system calls these widget methods for widget operation.

Test Data for Input Text

When widgets require text data, the tutorial system should show an example of the input. For this purpose, existing test data is used. In our method, we assume that the required test data is typical data for using the function. In this paper, we use the test data of JFCUnit [6]. JFCUnit is a software testing tool for Java Swing. The test data of JFCUnit is described as the correspondence a widget variable and a text datum. Therefore, the tutorial system can use the input text data by checking widget variables in sequence diagrams and in test data.

3.2 Added Tutorial System

The generated tutorial system is described using the AspectJ code. AspectJ is the one of the software tools that use aspect-oriented programming (AOP) [7]. To implement the tutorial system, the following two processes need to be added to the target software:

- Get the coordinates of widgets on the screen for showing the mouse pointer.
- Operate each widget, such as a mouse click and a text input

These two processes have to be added to each widget in the software. That is, they cannot be created as one class. They have to be added to many classes which process widget generation. Therefore, these two processes can be considered as crosscut concerns of AspectJ. Thus, these two processes are generated as AspectJ code in our method. It is possible to weave a tutorial system into target software without modifying the source code. Therefore, developers can easily maintain software.

4 System Architecture of ACTS

The system architecture of our system, called ACTS (Automatic Creation of Tutorial System), is shown in Figure 6. ACTS consists of the following five steps.

1. Adding extra description to sequence diagrams
2. Extracting function name from use case diagrams
3. Extracting operation information from sequence diagrams
4. Generating tutorial system automatically
5. Weaving tutorial system into target software automatically

4.1 Adding Extra Description to Sequence Diagrams

Target software developers output the use case diagrams and sequence diagrams described using various UML tools. Outputted data is written in the XMI format. ACTS creates a file in which required descriptions for generating a tutorial system are added to XMI-formatted sequence diagrams. The required descriptions are the widget variables, presentation sentences and test data for input text, and combined with the messages of sequence diagrams. Also, the window objects are specified in sequence diagrams. These extra descriptions are added by the target software developers. And ACTS adds test data of JFCUnit as input text to sequence diagrams.

4.2 Extracting Function Name from Use Case Diagrams

From the described use case diagrams, ACTS extracts the use case names of software functions used by the tutorial system. End users can watch an explanation of usage for a function by selecting the function name.

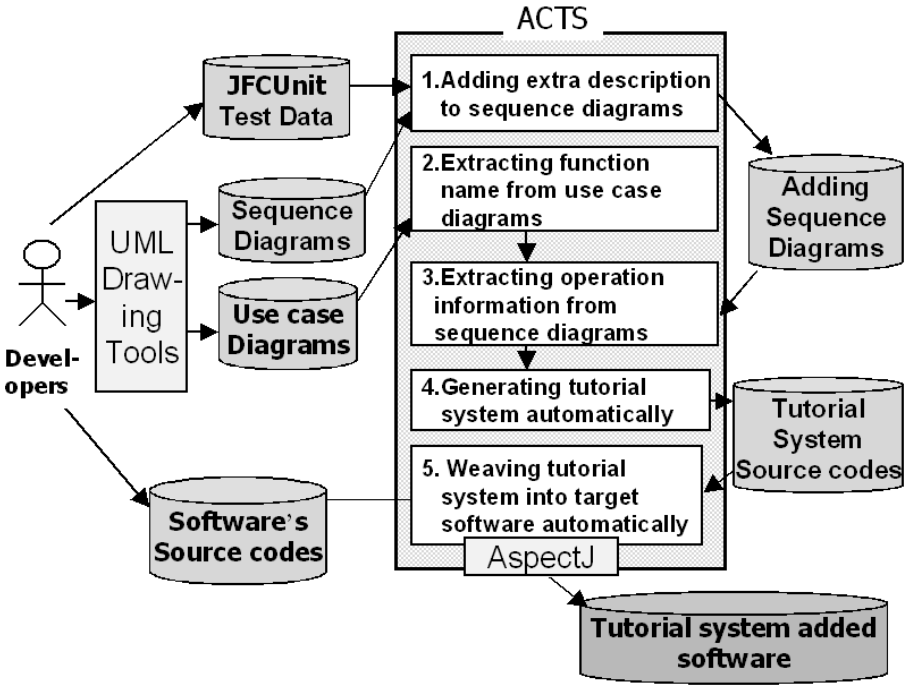


Fig. 6. System architecture of ACTS

4.3 Extracting Operation Information from Sequence Diagrams

ACTS extracts window object names and widget names from the described sequence diagrams. By using a widget name, the coordinates of each widget in a window can be calculated. A demonstration of mouse pointer movement based on the coordinates of each widget is performed. In our method, a mouse pointer is created as a visual image, and it is moved to each widget. Thus, our system can implement a demonstration of mouse pointer movement virtually.

The order in which the operations of the function are carried out is extracted from the flow of messages described in the sequence diagrams. By adding the explanation sentences of operations to the sequence diagrams, a suitable timing for displaying the explanation of an operation can be extracted.

4.4 Generating Tutorial System Automatically

A tutorial system is automatically generated by extracting function names from use case diagrams and information on how to operate the functions from sequence diagrams.

The procedure is shown as follows:

1. A process for displaying the function based on the extracted function name is generated for a tutorial.

2. An operation procedure based on the flow of the operation information in the messages of the sequence diagrams is constructed.
3. According to the constructed operation procedure, a process for the demonstration of mouse pointer movement from the location of the GUI widgets and end user operation is generated.
4. A process for displaying the explanation of the operation following the demonstration of mouse pointer movement is generated.

Generated source codes based on above the procedure include following contents.

- ACTS generates Java source codes creating a window. This window consists of buttons for representing function names of the target software and a text area for displaying the presentation sentences for the tutorial system.
- ACTS generates AspectJ source codes which demonstrate mouse pointer movement virtually based on the coordinates of widgets in the target software.
- ACTS generates Java source codes which display presentation sentences based on the operation procedure extracted from sequence diagrams.

4.5 Weaving Tutorial System into Target Software Automatically

Target software developers can add tutorial system to the target software by compiling generated Java source codes and weaving generated AspectJ source codes into the target software.

Widget names described in sequence diagrams are written in these AspectJ source codes. AspectJ software searches the target software for the same widget

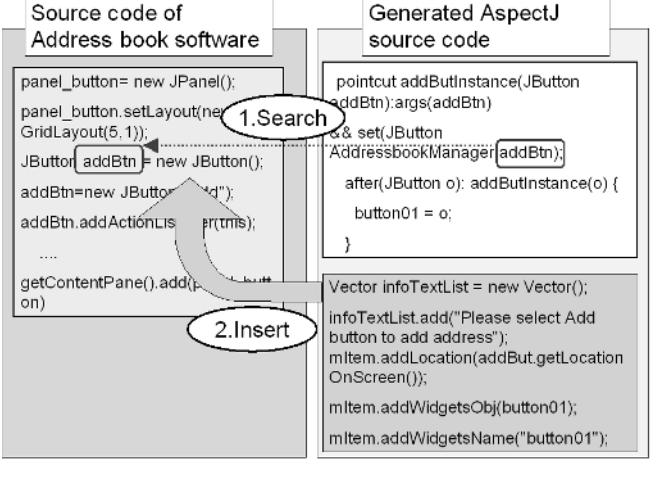


Fig. 7. A summary of clicking the button of "Add Address"

names as the widget names written in generated AspectJ source code, and weaves necessary codes into the target software, such as the procedure of getting the coordinates of widgets and displays the presentation sentences.

The summary of weaving generated AspectJ source code into target software is shown in Figure7. This is an example of clicking the button of "Add Address" in address book software.

In the example of address book software, AspectJ searches the widget name "addBtn" from the target software, because "addBtn" is written in the Pointcut of generated AspectJ source code(Figure7). AspectJ software adds following procedures to the found widget "addBtn" button.

- Get the coordinates of this button on the screen
- Show a mouse pointer on this button
- Highlight the presentation sentences when the current operation step of demonstration is about this button.

5 Evaluation

5.1 Add Some Extra Descriptions

Using the sequence diagrams of the following three types of software, we evaluate the burden placed on developers by the requirement of generating tutorial systems.

- Address book system software
- Mail-order system [8]
- Video rental system [9]

We counted the number of use cases, sequence diagrams, objects in each sequence diagram, and messages in each sequence diagram. The results are shown in Table 1. In this table, "Address" represents address book software, "Mail" represents mail-order system, and "Video" represents video rental system.

In our method, developers are required to add some extra descriptions to sequence diagrams. The required descriptions of sequence diagrams are the following three

- Specifying window objects
- Adding widget variables and explanation sentences of operations
- Adding text input examples for widgets

As shown in Table 1, "Window objects" indicates the number of specified window objects. "Extra descriptions" indicates the number of messages added as explanation sentences of operations and widget variables. "Extra text" indicates the number of messages added as examples of input text.

We verified that the rate of specified window objects per total number of objects is about 50% - 60%. The number of messages with extra descriptions added per total number of messages is about 50% - 60%. In these evaluations,

Table 1. Number of descriptions

	Address	Mail	Video
Use cases	4	15	7
Sequence diagrams	6	19	31
Objects	19	50	165
Window objects	13	28	95
Messages	53	102	379
Extra descriptions	32	51	138
Extra text	12	14	39

required extra descriptions for generating a tutorial system are of the same rate for other software tutorial systems.

Moreover, the number of messages with text input per total number of messages is about 10% - 20%. Input text is extracted from test data. Developers only select suitable test data. Therefore, developers are not required to create new input data for our method. The target software developers' burden does not become so big.

5.2 Correctness of Software Operation

Using the address book system software and video rental system, we evaluated whether the tutorial system generated by our method performs software operation correctly.

While executing the tutorial system, the following use cases were always performing correctly.

- "Add address data" in address book system software
- "Record new video" in video rental system
- "Record new cast" in video rental system

However, when the software did not have saved data, some use cases were not performed. For example, the following use cases were not performed.

- "Edit address data" in address book system software
- "Search video" in video rental system
- "Search cast" in video rental system

Therefore, when a use case needs some saved data, it is necessary to prepare the saved data for tutorial execution, or to specify the order in which tutorial execution is performed.

Next, we evaluate the correctness of performing the operations. The result is shown in Table 2. "Widgets" indicates the number of widgets, "Operating widgets" indicates the number of widgets operated by end users, "Performed operating widgets" indicates the number of widgets performing the operations correctly.

While executing the tutorial system, typical operations, such as clicking buttons (JButton), checking check boxes (JCheckBox) and selecting radio buttons (JRadioButton), were always performed correctly.

Table 2. Number of widgets and performed widgets

	Address	Video
Widgets	26	234
Operating widgets	14	77
Performed operating widgets	14	74

For the widgets that require input text data from end users, for example, text fields (JTextField) and text areas (JTextArea), the text data was automatically entered into the widget.

However, for the dialog boxes using JOptionPane, the tutorial system could not operate the widgets in the dialog boxes. The reason for this is as follows: Software with a GUI extended objects such as JFrame and JDialog often creates widgets in the constructor. That is, an instance of the extended object for displaying a GUI window on the screen is created.

At the same time, to display a dialog box on the screen an instance of JOptionPane is created. On JOptionPane, at least, some buttons are prepared as part of JOptionPane. The tutorial system cannot operate these prepared buttons on JOptionPane. Therefore, we should consider methods for operating JOptionPane.

6 Related Works

There are some existing methods for supporting the process of learning how to operate software. These include the user navigation system, the support system for learning how to operate software on the basis of operation logs, the GUI cover system, and other tutorial generation systems.

User Navigation System

The support system for learning how to operate software on the basis of operation logs takes the logs of user operations and uses them to understand the target function. This system analyzes the users' operations according to the goal. The next time the user uses the software, this system can show operations for that goal.

The following study is proposed for this method: development of a system which stores a log of a user's operations on a HCI (Human Computer Interaction) server, creates a model of the operations from this log, and then shows the operation [10].

The support system for learning how to operate software on the basis of recorded operation logs has the advantage of being able to allow user operation habits to be reflected in the next learning support session.

However, since it cannot determine which operation should be performed for the entire set of end users, this support system cannot perform an analysis using the end users operation log. Moreover, it is not necessarily that other users' operation logs are suitable for end user learning.

That is, this method is for users using the software many times; it is unsuitable as a learning aid for end users who use the software for the first time.

GUI Cover System

The GUI cover system [12] is a system that creates a new GUI screen which is different from the software's original GUI, and which makes operation possible on the new GUI. The new GUI screen is presented by hiding the software's original GUI; the new GUI screen, which functions as a substitute for the software's original GUI, link end user operations to software functions.

The GUI cover system involves preparing a GUI that can only be used for basic functions, even if it is for on advanced piece of software. The end users can change the GUI according to their preferences.

However, when a GUI cover is removed, it is noted as a disability that end users cannot operate the equivalent software function. That is, for end users who do not know how to use a new piece of software, a GUI cover system can reduce the burden on end users for learning the operations. However, when end users must use the software's original GUI, the learning effect is found to be inadequate.

Jedemo

Jedemo is a demonstration-authoring tool for Java applets [11].

In this method, developers add event-driven functions to a Java applet. The recorded event-driven functions operate the software automatically. Moreover, the event for automation is recordable from an operation. This method creates the help functions for Java applet software.

These are the advantages Jedemo:

- Since animated help functions can be created, new examination texts that software developers have to write are reduced.
- End users only need to push one button to see a demonstration of a concrete operation method.

Although these advantages do fulfill the aims of this method, the following points are noted as disadvantages. In order to generate a tutorial system, it is necessary to describe a new rule for introducing the concept of event-driven functions. In our method, a tutorial system is generated on the basis of use case diagrams and sequence diagrams, and these diagrams have already been described in the development stage. Therefore, our method imposes a smaller burden than this method.

7 Conclusions

In this paper, we propose a method for enabling developers to automatically weave a tutorial system into existing software, on the basis of use case diagrams,

sequence diagrams and test data. Using the tutorial system, end users can easily learn the operation of the software.

The following subjects remain for future work:

- Improvement of operation replication. A generated tutorial system in our method will support more GUI widgets.
- Supporting more test data formats. In this paper, at this moment, we use the test data for JFCUnit. We will support more GUI test tools.

References

1. G. Booch, I. Jacobson and J. Rumbaugh: "The Unified Modeling Language User Guide", Addison-Wesley (1998)
2. "aspectj Project", http://www.eclipse.org/aspectj/
3. "XML Metadata Interchange (XMI)": http://www.omg.org/technology/documents/formal/xmi.htm
4. T. Quatrani: "Visual Modeling With Rational Rose 2002 and UML", Addison-Wesley (2002)
5. "The IIOSS Project", http://www.iioss.org/index-e.html
6. "jfcUnit User Documentaiton": http://jfcunit.sourceforge.net/
7. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.M. Loingtier and J. Irwin: "Aspect Oriented Programming", PARC Technical Report, SPL97-008P9710042 (1997)
8. G. Schneider and J.P. Winters: "Applying Use Cases: A Practical Guide", Addison Wesley Longman, Inc. (2000)
9. J. Shirogane and Y. Fukazawa: "A Method of Scenario-Based GUI Prototype Generation", 3rd ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 02) (2002)
10. L.M. Encarnacao and S. Stoev: "An Application-Independent Intelligent User Support System exploiting Action-sequence based on User Modelling", Proc. of the Seventh International Conference on User Modeling (1999)
11. M. Miura and J. Tanaka: "Jedemo: Demonstrational Authoring Tool for Java Applets", Proc. of the Fifth World Conference on Integrated Design and Process Technology (IDPT2000) (2000)
12. H. Okada and T. Asahi: "GUI Navigator/Cover: GUI Transformation Systems for PC Novice Users", Proc. of the 10th International Conference on Human-Computer Interaction (HCI International 2003) (2003)

A Software Implementation Progress Model

Dwayne Towell[1] and Jason Denton[2]

[1] Abilene Christian University,
Abilene TX 79699, USA
dwayne.towell@acu.edu
[2] Texas Tech University,
Abilene TX 79602, USA
jason.denton@ttu.edu

Abstract. Software project managers use a variety of informal methods to track the progress of development and refine project schedules. Previous formal techniques have generally assumed a constant implementation pace. This is at odds with the experience and intuition of many project managers. We present a simple model for charting the pace of software development and helping managers understand the changing implementation pace of a project. The model was validated against data collected from the implementation of several large projects.

1 Introduction

Modern software development practices rely on periodically collected software metrics derived from a code base to provide management with feedback about the project and the process used to develop it [1, 2]. Well-defined and proven code metrics exist for some areas of software development [3, 4, 5], however the pace of implementation has no such established metrics based on code attributes. Several alternative, non-code-based progress metrics, such as function points [6] and earned value [1, 7], have been proposed and widely used. We believe it is possible to leverage existing size metrics to directly monitor progress in a code base, but to date such an approach has not been widely employed.

Here we propose implementation progress model based on development artifacts to interpret metrics and bridge the gap between concrete sampled data and expectations or beliefs about the underlaying process. On a small scale, this type of model may act as a predictor to set expectations over the next few data samples. This small-scale prediction helps provide timely feedback to management on the state of a project. Viewing a whole project, a model provides a portrait of the entire implementation process.

Without a formal, code-based implementation model management must rely on evidence other than implementation artifacts when making decisions about a project. In contrast, a formal implementation progress model based on implementation artifacts does not rely on external evidence yet establishes critical parameters and allows objectively evaluation based on inherent artifact attributes. Here we propose an implementation progress model based on implementation artifact metrics that matches our intuitive understanding of implementation progress.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 93–106, 2006.

The next section discusses supporting work. Section 3 describes the hypothesis, proposed implementation progress model, and research process. Results from actual projects are presented in Section 4. Finally, conclusions and directions for future study are in Section 5.

2 Related Work

Schneidewind uses time-series metrics to create a method for evaluating process stability [8]. Schneidewind asserts that metric trends are an indicator of the underlying process and that monitoring the trends can support managing the process. He suggests the shape of time-series data can be used to identify critical moments within a project. To further quantify project trends, an indirect metric based on time-series data is used. He defines a *change metric* as the difference between consecutive measurements of a primary metric. The model proposed here introduces a growth metric appropriate in the context of measuring project progress.

While discussing project progress, McConnell defines *code growth* for a project as the total size of project (source code) as a function of project time [2]. Code growth of traditional iterative development contains three distinct phases. In the first phase, architectural development and detailed design generate little code. The second phase provides staged deliveries and includes detailed design, coding, and unit testing. During this phase code growth is very high. Approaching initial delivery, the third phase, code growth slows to a crawl. Typical phase transitions occur at approximately 25% and 85% of the total implementation time for well-managed projects [2]. Specific details are not provided about metrics, but source lines of code, or a similar size metric, is assumed.

McConnell encourages the collection of time-series data to provide feedback supporting project management. Specifically, he recommends that collected data be viewed graphically since the *shape* can be used to diagnose project health. He graphically depicts the typical code growth pattern for a well-managed project, but acknowledges that its details varies to some degree. Our proposed progress model provides an empirical representation of the overall project shape and provides a specific interpretation of the three phases documented by McConnell.

3 Defining a Formal Progress Model

3.1 Informal Progress Models

Informal (non-mathematical) progress models already exist; as seen in project vocabulary and assumptions. Informal models are commonly used to answer project status queries, such as:

When will it be done, based on the current pace?

What was the size of the total effort for that project?

What fraction of the total effort has been spent?

Such informal progress models capture another key attribute of implementation progress. The informal model acknowledges that project speed is not constant throughout a project; projects "ramp up" and "slow down". These phrases refer to project speed and suggest the ability or desire to determine implementation velocity. As envisioned by experienced project managers, this velocity increases at the beginning and decreases near the end [2]. This is possibly an instance of the "S" shape progress or growth curse which is observed not only in projects but also in many other domains. A formal implementation progress model should be informed by this experience and capture the variations in velocity during implementation.

A formal implementation model should serve the same purpose as the informal model. The model must help answers questions about implementation speed and progress of current projects and provide a framework for making predictions about the future of the project. For example, changes in the rate of progress in an otherwise stable environment may indicate the project has transitioned to a new phase. This assumes the rate of progress is dependent on the project state.

3.2 Requirements for a Formal Progress Model

The interpretive power of an implementation progress model is important to consider. Interpretation of metric data relies on some understanding of our belief about the underlying process. In general, model parameters should be few in number, directly interpretable, and measured in existing units. These properties give the model parameters the most meaning and thus give the model the most explanatory power.

An implementation progress model should approximate actual project data collected. Figure 1 shows accumulated source lines of code sampled from the implementation phase of one project studied. The data in Figure 1 is very similar to the S-like curve described by McConnell [2].

This graph demonstrates the important characteristics of typical progress data. Overall progress is not linear with time; the fastest pace occurs during the middle of the project, while the ends are slower paced. The slope of the progress curve indicates the speed of progress.

3.3 Formal Implementation Progress Model

The primary goal of software implementation is creation of artifacts which contribute to delivering a working system. Implementation progress can be measured as change in an artifact. Progress over time can be measured as the sum of individual changes. We define implementation progress as the accumulated effort captured in code, which will eventually be delivered to the customer. For environments and development phases that emphasize code as the primary engineering delivery, we feel this is an appropriate definition.

Implementation metrics, traditionally used to measure size, can be employed to measure progress. Here we define a *growth metric* as the absolute difference between consecutive samples of a size metric, as shown in (1).

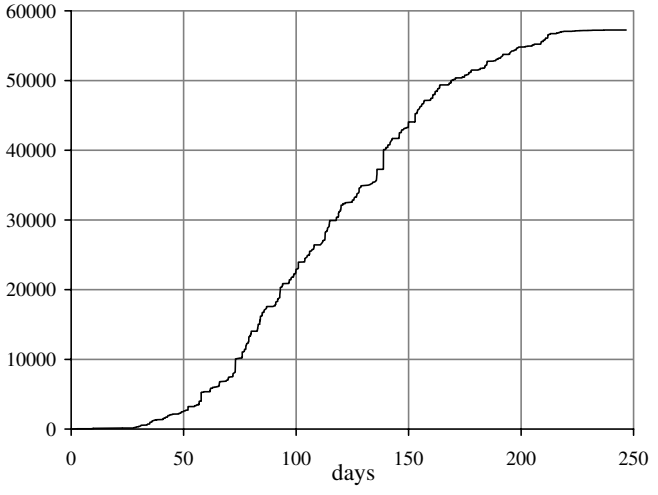


Fig. 1. Accumulated source lines of code changed for a sample project by day

$$\Delta_{m_t} = |m_t - m_{t-1}| \tag{1}$$

Where Δ_{m_t} is the growth in a metric m at time t.

Also useful is the idea of implementation velocity [9]; the rate at which progress is being made on a project. Evidence suggests implementation velocity begins and ends at zero while being at its highest in the middle. The simplest implementation velocity graph, consistent with experience, consists of three linear segments. Implementation velocity begins at zero. It increases linearly until the maximum sustainable velocity for implementation has been reached. The velocity remains constant until near the end of implementation when it begins to decrease. Then, implementation velocity constantly decreases until it reaches zero.

Figure 2 shows the idealized implementation velocity for a project as a function of time. The horizontal, center phase represents the steady, efficient development observed in the middle of the implementation phase. The positive slope at the left represents increasing velocity as implementation "gathers speed". The negative slope at the end of the graph shows the implementation phase decreasing speed as the end approaches.

The idealized graph shown here is symmetric; however, symmetry is not common in practice and is not required by the model presented. The idealized velocity as a function of time (v_t) can be described using three parameters.

$$v_t = \begin{cases} s\frac{t}{t_p}, & 0 \le t < t_p \\ s, & t_p \le t < t_q \\ s\frac{(t-t_f)}{t_q - t_f}, & t_q \le t \le t_f \end{cases} \tag{2}$$

In (2) the velocity is given as a function of time, where s is the maximum sustained velocity, t_p and t_q are the times of the phase transitions, and t_f is

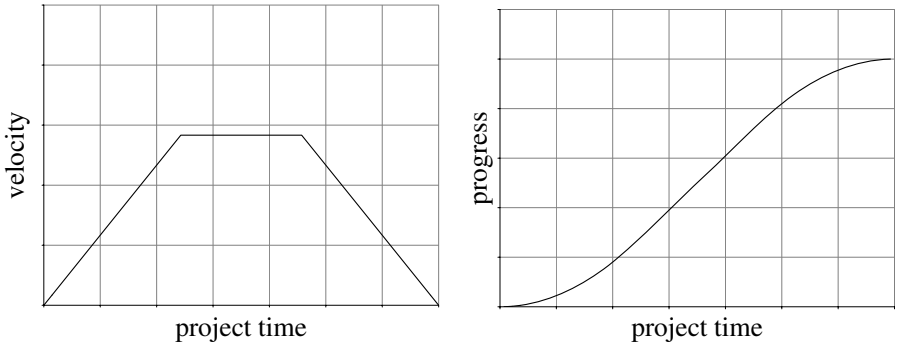


Fig. 2. Idealized implementation velocity as a function of time

the time at the end of implementation. Time may be measured in any real unit, such as days. Velocity is measured in size of metric change per time unit, such as lines of code per day.

Integration of the idealized velocity for a project produces idealized progress as a function of time (p_t).

$$p_t = \begin{cases} s\frac{t^2}{2t_p}, & 0 \le t < t_p \\ st - \frac{1}{2}st_p, & t_p \le t < t_q \\ s\frac{\left(t^2 - 2t_f t + t_q^2 + t_p t_f - t_p t_q\right)}{2(t_q - t_f)}, & t_q \le t \le t_f \end{cases} \tag{3}$$

The idealized implementation progress curve as a function of time is shown in (3). Progress is measured in accumulated metric growth to date, such as total lines of code changed.

4 Model Validation

We examined several size metrics as the basis for the growth metric used in our model [10]. *Source lines of code* (SLOC) is frequently used for estimating resources needed and should be readily available in most development environments [6, 11, 12, 13]. In this study, lines containing only white space and lines consisting of comment characters without any alphabetic characters were not counted. In addition, physical lines containing both code and comments were counted as two lines.

Two variations on the SLOC metric were considered. The simplest form counts the *SLOC change* (SLOCC) for each file. SLOCC is the absolute difference in SLOC between source files consecutively committed to the project repository; it counts SLOC added or deleted from the previous version. This assumes the correct removal of code artifacts is equivalent in terms of effort as correctly adding to the code base. We realize this may not strictly be the case, but it is difficult to determine what an appropriate weighting factor should be. To avoid introducing a weighting factor for this study, we assume all changes represent equal effort.

The second form measures the number of lines actually changed between submissions by comparing the files. This second measure is sometimes referred to as *code churn* (CHURN) [14]. CHURN is the count of source lines inserted, deleted, or changed between consecutively committed source files. It is probably a better change metric than SLOCC since CHURN captures more effort.

4.1 Alternative Models

Parameterized models provide an approximation of the sampled data for a particular data set. The model curve which most closely fits the data is considered the best; it introduces the least error. Model fit can be measured using the squared residual after subtracting the model curve from the sample data. To allow comparisons between models the average squared residual error ($\overline{R^2}$) is used. The model with the lowest $\overline{R^2}$ for a particular data set provides the closest approximation.

In addition to the proposed implementation progress model, three alternative models were chosen to provide a context for evaluating the fit of the proposed model.

The first model was a linear approximation. The linear model curve is given by (4). Linear approximation, with only two parameters, represents a practical lower-bound on the number of model parameters and the model with the highest expected $\overline{R^2}$.

$$linear_t = at + b \tag{4}$$

The second alternative model chosen was a multiphase, piecewise parabolic approximation. It contains eleven parameters; its model curve is shown in (5). This model was chosen to represent a practical lower-bound on $\overline{R^2}$.

$$multiphase_t = \begin{cases} at^2 + bt + c, & 0 \leq t < t_p \\ dt^2 + et + f, & t_p \leq t < t_q \\ gt^2 + ht + i, & t_q \leq t < t_f \end{cases} \tag{5}$$

The multiphase model was chosen to provide an highly data-conforming model. The proposed model is a special case of (5).

The third model was a third-degree polynomial approximation, with four parameters as shown in (6). A third-degree polynomial approximation provides just enough flexibility to model the S-curve observed. It also provides a model of approximately the same number of parameters as the proposed model.

$$polynomial_t = at^3 + bt^2 + ct + d \tag{6}$$

4.2 Experimental Data

Seventeen projects from a single company were studied. All projects were developed using the same iterative process. They were six weeks to eighteen months in length and involved one to eight engineers. All projects produced entertainment and education oriented software designed to be marketed to consumers for use

with Microsoft® Windows® and on Macintosh® personal computers between 1995 and 2002. In this environment, before the prevalence of the Internet, once this type of consumer product was released to manufacturing, no maintenance changes were possible due to economic considerations. Manufacturing and distribution costs meant the projects had clear delivery dates after which no work was to be done. This is unlike other environments, where software is delivered in near real-time or deployed, and implementation evolves into a continuous cycle of maintenance. The progress model studied is expected to be meaningful when applied to each release of on-going projects, however additional studies will be needed to establish this. We expect results from this homogeneous group of projects will apply to initial development efforts of iteratively developed projects and to projects without maintenance phases.

4.3 Model Fitting Results

Evaluations of both metrics for each project were performed. The three alternative models described above and the proposed model were used. A numerical fitting routine was used to find parameter values that minimized $\overline{R^2}$.

Figure 3 shows progress measured via accumulated SLOCC and model curves for a project. As expected, the linear model provides a poor fit for the data and the multiphase model fits the data very accurately. Both the polynomial and proposed models provide fits between the linear and multiphase models.

The polynomial model exhibits wild "swings" near the ends. These swings are typical of polynomial curves which tend to favor data points near the center rather than the ends. In this case, the polynomial model suggests a "negative" amount of accumulated work had been accomplished until about day forty of the project. Similarly, it indicates reverse-progress begins to occur around day 220. In almost all cases, these polynomial model swings suggest negative progress occurs at the beginning and end of the project.

The multiphase model includes discontinuities, occurring on day 72 and 138. These discontinuities represent an instantaneous change in speed, which is inconsistent with an intuitive understanding of the process. In general, small changes in a data set may radically change the location and size of the discontinuities, which suggests the model does not accurately represent the implementation process.

The average squared residual error ($\overline{R^2}$) for each model is given in the legend of Figure 3. The values agree with a visual assessment of the fit except in the case of the polynomial model. While the lower $\overline{R^2}$ for the polynomial model is more desirable, the polynomial fit suffers extensively from undesirable swings near the ends. These swings violate a basic expectation of accumulated progress, that is, it should be monotonically increasing. While the proposed model has a larger $\overline{R^2}$, it is monotonically increasing and behaves as expected. This behavior appears to support in-project predictions better than the polynomial model.

In this project, most of the proposed model $\overline{R^2}$ can be seen to occur in the first third of the project. Each of the other three metrics show similar results; this may indicate early efforts are not as efficiently captured by the metrics as later

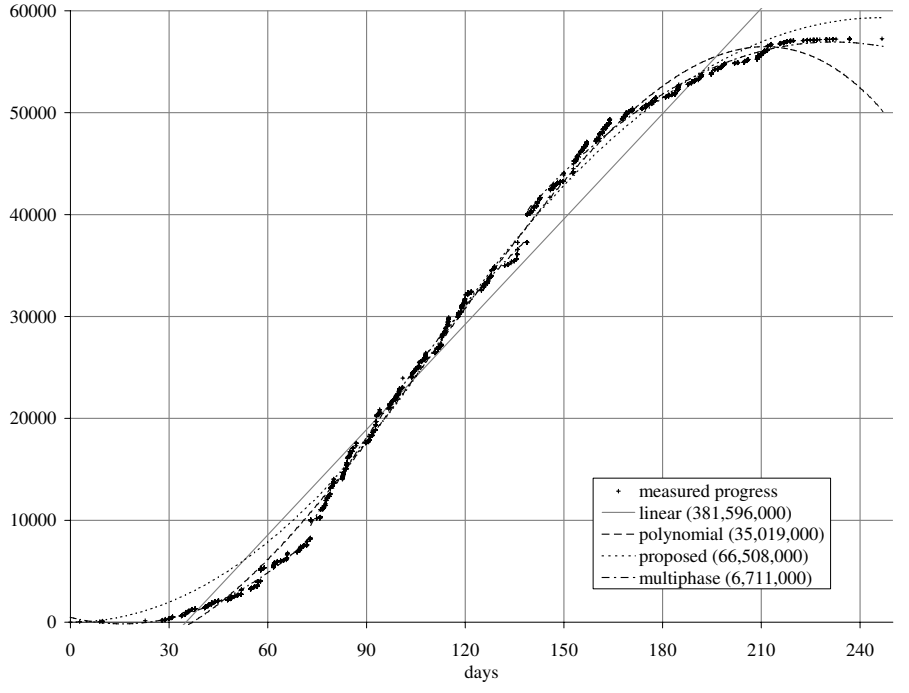


Fig. 3. Progress measured via accumulated source lines of code change (SLOCC) for project nine and progress model curves (with $\overline{R^2}$)

efforts. It could also indicate that during the later part of the project, the pace was unpredictably high (in violation of the implied model). Without additional information about the project, or its context, a determination cannot be made.

4.4 Data Analysis

The $\overline{R^2}$ for each metric is given in Tables 1 and 2. Figures 4 and 5 show $\overline{R^2}$ *relative* to the linear model $\overline{R^2}$ for each project. To improve viewing, projects are ordered by polynomial model relative $\overline{R^2}$.

In all cases the proposed model reduces $\overline{R^2}$ compared with the linear model, as expected, since the proposed model has an additional parameter. In many cases the proposed model *substantially* reduces $\overline{R^2}$ when compared with a linear model. In a few of these cases the reduction in $\overline{R^2}$ is almost to the level achieved using the multiphase model. In these conforming cases the proposed model provides a meaningful interpretation of the data.

Consider the proposed model $\overline{R^2}$ compared with the linear model $\overline{R^2}$. In cases where the proposed model substantially reduces $\overline{R^2}$, the model gave improved results with the addition of a single parameter. This substantial improvement suggests the data conforms to the model and the results may be relied upon to

Table 1. $\overline{R^2}$ measuring *source lines of code changed* (SLOCC)

		Model $\overline{R^2}$ (in millions)			
Project	samples	linear	polynomial	proposed	multiphase
1a	129	0.0667	0.0482	0.0434	0.0094
1b	1408	1.8269	1.6747	1.1458	0.4171
3	406	3.8133	0.9391	0.6371	0.0762
4	1394	1.0997	0.5958	0.6843	0.1176
5	90	0.0316	0.0108	0.0094	0.0021
6	1455	8.6590	2.0236	2.4551	0.2863
7	2204	12.3603	3.0171	4.0937	0.7440
8	138	0.0915	0.0245	0.0239	0.0028
9	1555	11.3106	1.1201	2.1672	0.2127
10	481	0.4575	0.0738	0.0386	0.0147
11	164	0.0111	0.0043	0.0037	0.0017
13	1274	2.4112	0.8349	0.8790	0.1589
14	715	0.4516	0.1139	0.1331	0.0181
15	723	0.1256	0.1215	0.1074	0.0210
17	827	2.9719	1.1158	1.2222	0.2215
19	967	6.3210	2.6441	1.7401	0.2475
20	1214	1.9231	0.3513	0.1566	0.0414

Table 2. $\overline{R^2}$ measuring *code churn* (CHURN)

		Model $\overline{R^2}$ (in millions)			
Project	samples	linear	polynomial	proposed	multiphase
1a	129	0.3485	0.2205	0.1601	0.0289
1b	1408	22.0988	8.2642	15.6206	1.7608
3	406	19.6692	4.9855	3.4455	0.2821
4	1394	2.6183	1.7401	2.0264	0.5516
5	90	0.2008	0.0530	0.0590	0.0119
6	1455	19.7174	5.3214	5.7359	0.8605
7	2204	36.1142	9.8627	12.7864	1.9951
8	138	0.2487	0.0674	0.0549	0.0087
9	1555	49.1829	4.7964	9.3587	0.8686
10	481	8.9162	2.8376	3.0023	0.0517
11	164	0.0362	0.0203	0.0155	0.0077
13	1274	9.4001	3.9973	3.2209	0.7597
14	715	1.7914	0.4591	0.5131	0.0593
15	723	0.4844	0.4684	0.4064	0.0814
17	827	11.0583	3.1825	3.3451	0.7040
19	967	20.5449	8.5899	5.9072	0.5758
20	1214	7.8927	1.7450	0.8902	0.1993

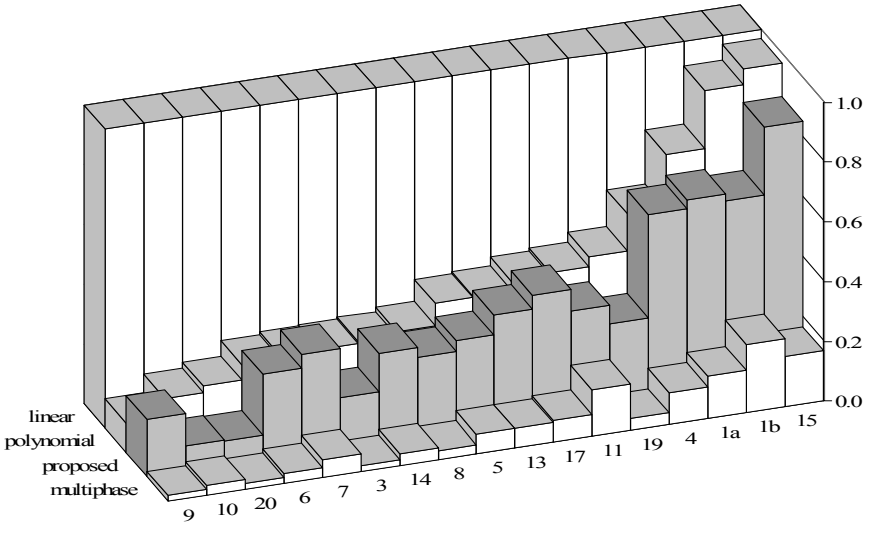


Fig. 4. *Source lines of code change* (SLOCC) average squared residual error ($\overline{R^2}$) relative to linear $\overline{R^2}$

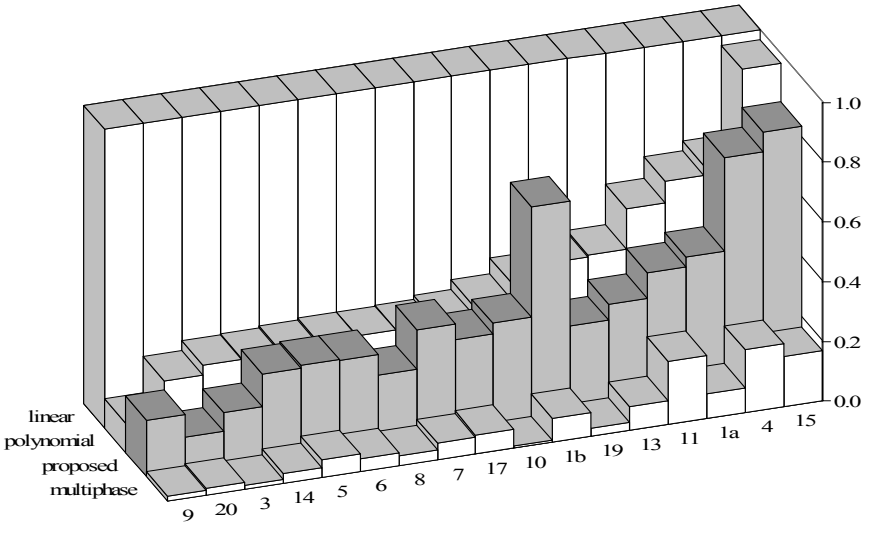


Fig. 5. *Code churn* (CHURN) average squared residual error ($\overline{R^2}$) relative to linear $\overline{R^2}$

correctly interpret the data. In the non-conforming cases, where the reduction is less significant, the model may not be appropriate and the results should only be used judiciously. Based on the available projects, we suggest the proposed model may be relied upon when the $\overline{R^2}$ is at most half that of the linear model.

Projects may fail to conform to the model for a number of reasons. Almost all projects exhibit "pauses" corresponding with weekends when developers do no work. Some projects also include larger periods when no apparent progress is made during a holiday break. Both of these phenomena can be seen clearly in projects 1a and 15. Using work days instead of calendar days would eliminate a major cause of time-related noise. Several projects include substantial, sudden, and anomalous progress. In all cases where these events were examined closely, the anomaly has proven to be the result of an unfortunate side-effect of the specific data collection procedure used. For example, a renamed file was detected as a combination of a substantial deletion and a subsequent addition. A commitment to collect the needed data during the project could reduce noise by allowing anomalies to be detected and corrected while any additional required information is still available.

Tables 3 and 4 show the model parameters for each project and metric, ordered by $\overline{R^2}$ relative to linear $\overline{R^2}$. With only seventeen projects and no independent data available, few definite conclusions can be reached, however several items are worth noting.

In about half of all cases, the model indicates t_p and t_q are essentially the same. In these cases, the model $t_q - t_p$ is close to zero, suggesting steady progress did not occur; implementation was either accelerating or decelerating. This could indicate development under a tight schedule or a process that could be improved. It is also interesting to note that in these cases the model was able to substantially reduce $\overline{R^2}$ while effectively using only two parameters.

Table 3. *Source lines of code change* (SLOCC) progress model parameters and $\overline{R^2}$ relative to linear $\overline{R^2}$

		Model parameters			
Project	relative $\overline{R^2}$	s	t_p	t_q	t_f
20	0.081	289.8	64.2	64.3	187.1
10	0.084	156.9	30.6	30.7	132.0
3	0.167	376.0	48.9	49.0	120.3
9	0.192	480.4	109.8	109.8	246.7
8	0.262	42.5	17.2	17.5	128.9
19	0.275	501.7	108.1	108.4	164.8
6	0.284	361.9	57.5	126.1	248.0
14	0.295	91.8	21.3	97.8	238.2
5	0.297	92.0	14.9	14.9	48.8
7	0.331	183.9	93.4	288.9	555.2
11	0.336	48.9	27.5	62.4	70.9
13	0.365	250.5	93.8	213.9	220.8
17	0.411	268.4	6.3	110.4	181.0
4	0.622	215.6	24.1	138.5	201.7
1b	0.627	303.1	28.9	220.2	248.3
1a	0.650	183.8	14.0	14.1	44.9
15	0.855	92.3	4.5	140.9	163.9

Table 4. *Code churn* (CHURN) progress model parameters and $\overline{R^2}$ relative to linear $\overline{R^2}$

		Model parameters			
Project	relative $\overline{R^2}$	s	t_p	t_q	t_f
20	0.113	560.6	68.2	68.3	187.1
3	0.175	820.9	45.7	45.7	120.3
9	0.190	1028.0	110.6	110.7	246.7
8	0.221	75.9	21.5	21.8	128.9
14	0.286	170.6	23.2	92.4	238.2
19	0.288	882.7	111.8	112.0	164.8
6	0.291	631.7	43.4	136.4	248.0
5	0.294	211.3	17.5	21.4	48.8
17	0.302	538.3	10.7	109.3	181.0
10	0.337	412.6	16.7	16.8	132.0
13	0.343	548.8	87.9	211.6	220.8
7	0.354	325.9	85.2	298.6	555.2
11	0.427	100.9	20.4	59.2	70.9
1a	0.459	365.8	14.5	14.5	44.9
1b	0.707	667.8	54.1	238.3	248.3
4	0.774	386.7	17.5	148.2	201.7
15	0.839	181.7	0.5	139.7	163.9

In conforming cases where $t_q - t_p$ is much larger than zero, the model indicates steady, sustained implementation occurred between t_p and t_q. In these cases, the implementation velocity (s) can be stated with great confidence. Velocity is a surrogate for productivity in the dimension measured by the specific metric. For example, Table 3 shows project six averaged over 360 lines of new code per calendar day between project days 58 and 126.

In the projects studied, implementation velocity (s) varies by more than an order of magnitude. While part of this variation is due to the number of engineers assigned to the project, likely some is due to proficiency. This is consistent with studies showing individual programmer productivity varies by as much as an order of magnitude [15].

5 Conclusions

Interpreting implementation progress measurements is difficult. A simple model is needed to provide a framework to help interpret the data. We have developed a piecewise approximation based on a three-phase model of linear implementation velocity. The model corresponds well to our intuition of how project progress occurs. It identifies project phase boundaries as well as the velocity of implementation during each phase. Furthermore, the progress model allows comparisons of project velocity between projects and easily supports estimating.

The progress model fits the available sample data better than a linear model. With only one additional parameter, the model produces fits with approximately

two-thirds less error than a linear fit. When compared with a polynomial fit, the progress model performs at least as well as a polynomial model which has one additional parameter.

Any model is only as good as the data on which it is based. Errors were discovered in both dimensions of the sample data. Spurious data entries were occasionally introduced due to the check-in process used. Similarly, using project work days, instead of calendar days, could have improved the quality of data in the time dimension.

5.1 Future Work

This work provides a sound basis for further study in this area. The progress model presented here only considers non-maintenance implementation. Projects with clear delivery dates, after which continuing development is not planned, fall into this category. Projects in maintenance or under continuous development may not exhibit phases similar to projects with firm end dates and deserve to be investigated, although this would require further work.

The stability of the model suggests it could be used to make predictions. Estimating project parameters such as final size, delivery date, development pace, etc. during implementation should be investigated. Similarly, comparisons of teams or projects based on model parameters could be studied.

Investigation of other metrics as a basis for measuring progress should be undertaken. If a size metric for object-oriented software were developed, investigating its use as a basis for a growth metric would be very valuable. Variations of existing metrics better tuned to capture change should be studied. One example of this type of metric is the sum of cyclomatic complexity of all changed functions, rather than simply the change in cyclomatic complexity of a source artifact.

References

1. Boehm, B.W.: Software Engineering Economics. Prentice-Hall (1981)
2. McConnell, S.: Software Project Survival Guide. Microsoft Press, Redmond, WA (1998)
3. Fenton, N.E.: Software Metrics: A Rigorous Approach. Chapman and Hall, London (1991)
4. Kafura, D., Canning, J.: A validation of software metrics using many metrics and two resources. In: Proceedings of the 8th International Conference on Software Engineering. (1985) 378–385
5. Fenton, N.E., Neil, M.: Software metrics: roadmap. In: Proceedings of the conference on The future of Software Engineering, ACM Press (2000) 357–370
6. Albrecht, A.J., John E. Gaffney, J.: Software function, source lines of code, and development effort prediction: A software science validation. IEEE Transactions on Software Engineering **9**(6) (1983) 639–648
7. Humphrey, W.S.: A Discipline for Software Engineering. Addison-Wesley (1994)
8. Schneidewind, N.F.: Measuring and evaluating maintenance process using reliability, risk, and test metrics. IEEE Transactions on Software Engineering **25**(6) (1999) 761–781

9. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (1999)
10. Towell, D.: An implementation progress model. Master's thesis, Texas Tech University (2004)
11. DeMarco, T.: Controlling Software Projects - Management, Measurement and Estimation. Yourdon Press, Inglewood Cliffs, NJ (1982)
12. Lind, R.K., Vairavan, K.: An experiemental investigation of software metrics and their relationship to software development effort. IEEE Transactions on Software Engineering **15**(5) (1989) 649–653
13. Jorgensen, M.: Experience with the accuracy of software maintenance task effort prediction models. IEEE Transactions on Software Engineering **21**(8) (1995) 674–681
14. El-Eman, K.: A methodology for validating software product metrics. Technical Report NRC/ERB-1076 44142, National Research Council Canada, Institute for Information Technology (2000)
15. H. Sackman, W. J. Erikson, E.E.G.: Exploratory experimentation studies comparing online and offline programming performance. Communications of the ACM **1**(1) (1968) 3–11

Regular Inference for State Machines with Parameters*

Therese Berg[1], Bengt Jonsson[1], and Harald Raffelt[2]

[1] Department of Computer Systems, Uppsala University, Sweden
{thereseb, bengt}@it.uu.se
[2] Chair of Programming Systems and Compiler Construction,
University of Dortmund, Germany
harald.raffelt@cs.uni-dortmund.de

Abstract. Techniques for inferring a regular language, in the form of a finite automaton, from a sufficiently large sample of accepted and nonaccepted input words, have been employed to construct models of software and hardware systems, for use, e.g., in test case generation. We intend to adapt these techniques to construct state machine models of entities of communication protocols. The alphabet of such state machines can be very large, since a symbol typically consists of a protocol data unit type with a number of parameters, each of which can assume many values. In typical algorithms for regular inference, the number of needed input words grows with the size of the alphabet and the size of the minimal DFA accepting the language. We therefore modify such an algorithm (Angluin's algorithm) so that its complexity grows not with the size of the alphabet, but only with the size of a certain symbolic representation of the DFA. The main new idea is to infer, for each state, a partitioning of input symbols into equivalence classes, under the hypothesis that all input symbols in an equivalence class have the same effect on the state machine. Whenever such a hypothesis is disproved, equivalence classes are refined. We show that our modification retains the good properties of Angluin's original algorithm, but that its complexity grows with the size of our symbolic DFA representation rather than with the size of the alphabet. We have implemented the algorithm; experiments on synthesized examples are consistent with these complexity results.

1 Introduction

Model-based techniques for verification and validation of reactive systems, such as model checking and model-based test generation [1] have witnessed drastic advances in the last decades. They depend on the availability of a model, specifying the intended behavior of a system or component, which typically is developed during specification and design. However, in practice often no formal specification is available, or becomes outdated as the system evolves over time. In, e.g., the telecommunication area, revision cycles are extremely short, and at the

* Supported in part by the Swedish Research Council, and by the FP6 Network of Excellence ARTIST2.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 107–121, 2006.

same time the short revision cycles necessitate extensive testing and verification. Therefore, there are many cases where the only means to attain correspondence between model and system component is to construct a model directly from the component. Such models can be constructed by static analysis techniques using its source code, as in software verification (e.g., [2, 3, 4, 5]). However, many system components, including peripheral hardware components, library modules, or third-party components do not allow static analysis of source code, implying that models must be constructed from observations of their external behavior.

The construction of models from observations of component behavior can be performed using techniques for regular inference. Such techniques have been used, e.g., to create models of environment constraints with respect to which a component should be verified, for regression testing to create a specification and a test suite [6, 7], to perform model checking without access to code or to formal models [8, 9], for program analysis [10], and for formal specification and verification [11]. For finite-state reactive systems, the regular inference problem means to infer a regular language (in the form of a deterministic finite automaton) from the answers to a finite set of *membership queries*, each of which asks whether a certain word is accepted by the system component under test (SUT) or not. There are several techniques (e.g., [12, 13, 14, 15, 16, 17, 18]) which use essentially the same basic principles. Given "enough" membership queries, the constructed automaton will be a correct model of the SUT. Angluin [12] and others introduce *equivalence queries* which check whether the regular inference procedure is completed; if not they are answered by a counterexample on which the current hypothesis and the SUT disagree.

We intend to use regular inference to construct models of communication protocol entities. Such entities typically communicate by messages that consists of a protocol data unit (PDU) type with a number of parameters, each of which can assume several values. The alphabet of such models is thus typically very large. Since existing algorithms for regular inference use a number of queries, which grows polynomially with the size of the alphabet, they are not well suited for this situation. If some PDU parameters are irrelevant or almost never used, the algorithm should not be disturbed by their presence.

In this paper, we modify an algorithm for inferring a regular language, so that it is better adapted for inferring system components with large alphabets that are built from a small set of action types, each of which has a number of parameters. Most of these algorithms are based on similar principles: we choose Angluin's algorithm [12] since it is well known, and since we have an existing implementation for this algorithm [19]. The problem of inferring state machines where messages have arbitrary parameters appears to be very challenging. As a first step, we will in this paper assume that all parameters are booleans, and that a SUT can be modeled as an automaton, in which each transition is labeled by a PDU type and a guard over its parameters. We assume that guards are conjunctions over positive and negated parameter values. Furthermore, we will not consider the problem of inferring parameters of possible output data, but only how input parameters affect the state changes of a state machine. Ideas for

how to extend these rather restrictive limitations are sketched in the last section of the paper.

Algorithms for regular inference must represent the inferred automaton in terms of externally observable elements. A state is represented by a set $[u]$ of input words u such that the automaton after reading u reaches this state. For each input symbol a, the transition from $[u]$ for input a is constructed by determining which state is reached after reading ua. In the parameterized case, input symbols are of the form $\alpha(d_1, \ldots, d_n)$, where α is an action type and $d_1, \ldots, d_n$ is a tuple of boolean parameter values. We could naively use Angluin's algorithm to find the state reached after each of these 2^n different input symbols. Instead, we will strive to save work by assuming that from each automaton state, many of the input symbols have the same effect on the SUT, and can be regarded as equivalent. We can then construct a symbolic automaton representation, where the effect of each set of equivalent input symbols is represented by a transition from this state, labeled by a guard, i.e., a boolean expression over the parameters, which characterizes the equivalence class. In cases where the number of equivalence classes is small, we would like to perform the inference with less work (as measured by the number of membership queries) than by a naive application of Angluin's algorithm.

Our inference algorithm maintains, for each inferred state, a partitioning of subsequent input symbols into assumed equivalence classes. Each class is represented by a small set of representative input symbols that (as far as we have observed) have the same effect on the SUT. If later, new information is obtained which contradicts this assumption, the equivalence class is split, thus also splitting transitions and generating more refined guards. The guard that labels a transition is obtained by a search procedure to identify precisely the effect of parameter values, inspired by work on learning of conjunctions, e.g., [16, Ch. 1.3].

In order to develop a consistent algorithm to do the above, we present in this paper two significant extensions of Angluin's algorithm:

1. We generalize Angluin's algorithm so that it can infer a "partially defined" automaton, which from each state defines the effect of a set of representative input symbols. The representative symbols are in general only a subset of all input symbols.
2. We define a mechanism for inferring guards of a parameterized system from the symbols in an underlying partially defined automaton, by replacing the representative symbols by guards that characterize the transitions represented by each symbol. Extra queries may need to be performed to determine guards more precisely.

Our resulting inference algorithm is intended to infer parameterized systems where guards of transitions use only a small subset of all parameters of a particular action type. We establish an upper bound on the number of posed membership queries, which is exponential in the number of parameters that appear in guards. In contrast, using Angluin's original algorithm requires a number of membership queries which is exponential in the total number of parameters of

input symbols. On the other hand, the number of equivalence queries may grow in our case, since we add possibilities to construct hypothesized automata based on less information than in the original algorithm. We have performed a set of experiments on synthesized examples, which confirm this picture.

Organization. The paper is organized as follows. In the next section, we review Angluin's algorithm for inferring regular sets, and present a modification which can cope with the situation that the queries investigate different sets of suffixes for different prefixes. In Section 3, we present parameterized systems, and the technique to learn "partially defined" automata, from which guards of transitions are inferred. We prove that our algorithm retains good properties of Angluin's original algorithm, and establish upper bounds on the number of performed queries. Section 4 describes how we have implemented the ideas of the preceding section, and Section 5 presents the outcome of experiments on synthesized examples. Conclusions are presented in Section 6.

2 Inference of Finite Automata

In this section, we review the ideas underlying Angluin's algorithm, and present our generalization.

Let Σ be a finite alphabet of symbols. A *deterministic finite automaton* (*DFA*) over Σ is a structure $\mathcal{M} = (Q, \delta, q_0, F)$ where Q is a non-empty finite set of *states*, $q_0 \in Q$ is the *initial state*, $\delta : Q \times \Sigma \to Q$ is the *transition function*, and $F \subseteq Q$ is the set of *accepting states*. The transition function is extended from input symbols to words of input symbols in the standard way, by defining

$$\begin{aligned} \delta(q, \varepsilon) &= q \\ \delta(q, ua) &= \delta(\delta(q, u), a) \end{aligned}$$

An input word u is *accepted* iff $\delta(q_0, u) \in F$. The *language* accepted by $\mathcal{M}$, denoted by $\mathcal{L}(\mathcal{M})$, is the set of accepted input words.

Angluin's algorithm is designed to infer a (minimized) DFA $\mathcal{M}$ from a set of queries, each of which reveals whether a certain word is accepted or not. The algorithm is formulated in a setting, where a so called *Learner*, who initially knows nothing about $\mathcal{M}$, is trying to infer $\mathcal{M}$ by asking queries, which are of two kinds.

- A *membership query* consists in asking whether a word $w \in \Sigma^*$ is in $\mathcal{L}(\mathcal{M})$.
- An *equivalence query* consists in asking whether a hypothesized DFA $\mathcal{H}$ is correct, i.e., whether $\mathcal{L}(\mathcal{H}) = \mathcal{L}(\mathcal{M})$. The *Oracle* will answer *yes* if $\mathcal{H}$ is correct, or else supply a *counterexample*, which is a word u that is either in $\mathcal{L}(\mathcal{M}) \setminus \mathcal{L}(\mathcal{H})$ or in $\mathcal{L}(\mathcal{H}) \setminus \mathcal{L}(\mathcal{M})$.

The typical behavior of a *Learner* is to start by asking a sequence of membership queries, and gradually build a hypothesized DFA $\mathcal{H}$ using the obtained answers. When the *Learner* feels that she has built a "stable" hypothesis $\mathcal{H}$, she makes

an equivalence query to find out whether $\mathcal{H}$ is equivalent to $\mathcal{M}$. If the result is successful, the *Learner* has succeeded, otherwise she uses the returned counterexample to revise $\mathcal{H}$ and perform subsequent membership queries until converging at a new hypothesized DFA, etc.

Let us represent the information gained by the *Learner* at any point during the learning process, as a partial mapping Obs from Σ^* to $\{+, -\}$, where $+$ stands for *accepted* and $-$ for *rejected*. The domain $Dom(Obs)$ of Obs is the set of words for which membership queries have been performed, or which the *Oracle* has given as counterexamples in equivalence queries. An inference algorithm should prescribe how to transform Obs into a DFA $\mathcal{H} = (Q, \delta, q_0, F)$, which is *conformant* with Obs, in the sense that any word $u \in Dom(Obs)$ is accepted by $\mathcal{H}$ if $Obs(u) = +$ and rejected by $\mathcal{H}$ if $Obs(u) = -$. In general, there are many such automata, and the problem to find a smallest (in number of states) such automaton is NP-complete [20]. Angluin and others circumvent this problem by prescribing conditions on $Dom(Obs)$, under which it is "easy" to find a unique smallest automaton. These conditions regard each word in $Dom(Obs)$ as the concatenation of a prefix and a suffix. The idea is that prefixes are candidates for representing states of the hypothesized automaton, whereas suffixes are used to distinguish the states.

Angluin [12] supports this prefix-suffix view by representing Obs in terms of an *observation table* $\mathcal{T}$, which is a partial function from a prefix-closed set $Dom(\mathcal{T}) \subseteq \Sigma^*$ of prefixes. For each $u \in Dom(\mathcal{T})$, $\mathcal{T}(u)$ is a partial function from a set $Dom(\mathcal{T}(u)) \subseteq \Sigma^*$ of suffixes to $\{+, -\}$. It is required that $\varepsilon \in Dom(\mathcal{T}(u))$ for each $u \in Dom(\mathcal{T})$. We write $Entries(\mathcal{T})$ to denote $\{(u, v) : u \in Dom(\mathcal{T})$ and $v \in Dom(\mathcal{T}(u))\}$. An observation table $\mathcal{T}$ represents the partial mapping Obs if $uv \in Dom(Obs)$ and $Obs(uv) = \mathcal{T}(u)(v)$ whenever $(u, v) \in Entries(\mathcal{T})$.

Define the *short prefixes* of an observation table $\mathcal{T}$, denoted $Sp(\mathcal{T})$, to be the set of words $u \in Dom(\mathcal{T})$ such that $ua \in Dom(\mathcal{T})$ for some $a \in \Sigma$. An observation table $\mathcal{T}$ is *complete* if $ua \in Dom(\mathcal{T})$ for all $u \in Sp(\mathcal{T})$ and $a \in \Sigma$; it is *suffix-closed* if $(u, av) \in Entries(\mathcal{T})$ where $u \in Sp(\mathcal{T})$ and $a \in \Sigma$ implies that $(ua, v) \in Entries(\mathcal{T})$. For $u, u' \in Dom(\mathcal{T})$, let $u \approx_{\mathcal{T}} u'$ denote that $\mathcal{T}(u)(v) = \mathcal{T}(u')(v)$ whenever $v \in (Dom(\mathcal{T}(u)) \cap Dom(\mathcal{T}(u')))$. The table $\mathcal{T}$ *partitioned* if $\approx_{\mathcal{T}}$ is an equivalence relation on $Dom(\mathcal{T}(u))$. A partitioned table is *closed* if whenever $(u, v) \in Entries(\mathcal{T})$ there is a $u' \in Sp(\mathcal{T})$ with $u \approx_{\mathcal{T}} u'$ and $v \in Dom(\mathcal{T}(u'))$; it is *consistent* if $ua \approx_{\mathcal{T}} u'a$ whenever $ua, u'a \in Dom(\mathcal{T})$ and $u \approx_{\mathcal{T}} u'$.

Angluin showed how to construct a unique minimal automaton from a complete, closed, and consistent observation table in the case that $Dom(\mathcal{T}(u))$ is the same for all $u \in Dom(\mathcal{T})$. Our goal in this section is to generalize this construction to the case where the set $Dom(\mathcal{T}(u))$ of suffixes may differ significantly for different prefixes $u \in Dom(\mathcal{T})$.

Definition 1. *Let $\mathcal{T}$ be a partitioned, complete, closed, and consistent observation table. Define the DFA $\mathcal{T}/\approx_{\mathcal{T}}$ as (Q, δ, q_0, F), where*

- $Q = Dom(\mathcal{T})/\approx_{\mathcal{T}}$, *i.e.,* Q *is the set of equivalence classes of* $\approx_{\mathcal{T}}$,
- $\delta([u], a) = [ua]$ *for* $u \in Sp(\mathcal{T})$,
- $q_0 = [\varepsilon]$,
- $F = \{[u] \ : \ \mathcal{T}(u)(\varepsilon) = +\}$ □

Note how closedness and completeness ensures that we can define a transition for each equivalence class and symbol in Σ, and how consistency ensures that such transitions have a unique target equivalence class.

We are now ready to state a general theorem that gives constraints on any FSM that is conformant with an observation function.

Theorem 1 (Characterization Theorem). *Let* $\mathcal{T}$ *be a partitioned, complete, closed, and consistent observation table which represents Obs. If* $\mathcal{T}$ *is suffix-closed, then the DFA* $\mathcal{T}/\approx_{\mathcal{T}}$ *is the minimal automaton conformant with Obs.*

Angluin's algorithm uses a specialization of the conditions in Theorem 1, where $Dom(\mathcal{T}(u))$ is the same for all $u \in Dom(\mathcal{T})$.

3 Inference of Parameterized Systems

In this section, we consider how to adapt the techniques of the previous section to a setting where symbols in the alphabet are messages with parameters, e.g., as in a typical communication protocol. Since the problem of inferring state machines where messages have arbitrary parameters appears to be very challenging, we will here assume that all parameters are booleans, and that a SUT can be modeled as an automaton, in which each transition is labeled by a PDU type and a guard over its parameters. We assume that guards are conjunctions over positive and negated parameter values. Furthermore, we will not consider the problem of inferring parameters of possible output data, but only how input parameters affect the state changes of a state machine.

Let *Act* be a finite set of *actions*, each of which has a nonnegative arity. Let Σ_{Act} be the set of *symbols* of form $\alpha(d_1, \ldots, d_n)$, where α is an action of arity n, and $d_1, \ldots, d_n$ are booleans. We will use 0 and 1 to denote the boolean values *false* and *true*, respectively.

We assume a set of *formal parameters*, ranged over by $p, p_1, p_2, \ldots$. A *parameterized action* is a term of form $\alpha(p_1, \ldots, p_n)$, where α is an action α of arity n, and $p_1, \ldots, p_n$ are formal parameters. A *guard* for $\alpha(p_1, \ldots, p_n)$ is a conjunction whose conjuncts are of form p_i or $\neg p_i$ with $p_i \in \{p_1, \ldots, p_n\}$. We write $\overline{p}$ for $p_1, \ldots, p_n$ and $\overline{d}$ for $d_1, \ldots, d_n$. A *guarded action* is a pair $(\alpha(\overline{p}), g)$, where $\alpha(\overline{p})$ is a parameterized action, and g is a *guard* for $\alpha(\overline{p})$. A guarded action $(\alpha(\overline{p}), g)$ denotes the set $[\![(\alpha(\overline{p}), g)]\!] = \{\alpha(\overline{d}) \ : \ g[\overline{d}/\overline{p}]\}$ of symbols, whose parameters satisfy g.

Definition 2 (Parameterized system). *Let Act be a finite set of actions. A* parameterized system *over Act is a tuple* $\mathcal{P} = (Q, \longrightarrow, q_0, F)$, *where*

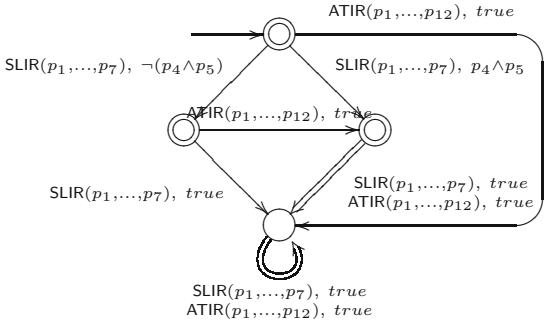


Fig. 1. Example of a parameterized system

- Q *is a finite set of states,*
- $\longrightarrow$ *is a finite set of* transitions. *Each transition is a tuple* $\langle q, \alpha(\overline{p}), g, q' \rangle$, *where* $q, q' \in Q$ *are states, and* $(\alpha(\overline{p}), g)$ *is a guarded action,*
- $q_0 \in Q$ *is the initial state, and*
- $F \subseteq Q$ *is a set of accepting states,*

which is completely specified and deterministic, i.e., for each state q *and symbol* $\alpha(\overline{d})$, *there is exactly one transition* $\langle q, \alpha(\overline{p}), g, q' \rangle$ *from* q *such that* $\alpha(\overline{d}) \in [\![(\alpha(\overline{p}), g)]\!]$. □

We write $q \xrightarrow{\alpha(\overline{p}),g} q'$ to denote that $\langle q, \alpha(\overline{p}), g, q' \rangle \in \longrightarrow$. A parameterized system is *expanded* if whenever $q \xrightarrow{\alpha(\overline{p}),g'} q'$ and $q \xrightarrow{\alpha(\overline{p}),g''} q''$, and in addition p_i or $\neg p_i$ is a conjunct of g', then either p_i or $\neg p_i$ must be a conjunct of g''. In other words, a parameterized system is expanded if all transitions from a state for some action test the same set of parameters. In Fig. 1 a fragment of a protocol provided by Mobile Arts AB [21] is given as an example of a parameterized system.

A parameterized system $\mathcal{P} = (Q, \longrightarrow, q_0, F)$ over Act denotes the DFA $\mathcal{M}_\mathcal{P} = (Q, \delta, q_0, F)$ over Σ_{Act}, where δ is defined by

$$\delta(q, \alpha(\overline{d})) = q' \qquad \text{whenever} \qquad q \xrightarrow{\alpha(\overline{p}),g} q' \text{ and } \alpha(\overline{d}) \in [\![(\alpha(\overline{p}), g)]\!].$$

Note that δ is well-defined, since $\mathcal{P}$ is completely specified and deterministic.

We will adapt Angluin's algorithm to inference of parameterized systems, in a situation where each symbol typically has many parameters, but for which the number of outgoing transitions from each state is small compared to the number of symbols in Σ_{Act}. Ideally, the effort needed to learn a parameterized system $\mathcal{P}$ should be in proportion to the size of its description as a parameterized system, and not to its number of states and $|\Sigma_{Act}|$, as is the case for Angluin's algorithm.

To accomplish this, we make two extensions to Angluin's algorithm. First, we must abandon the requirement that the constructed observation table $\mathcal{T}$ be complete, since then $Dom(\mathcal{T})$ is at least $|\Sigma_{Act}|$ times larger than the number of states of the constructed automaton. Instead of requiring that $\mathcal{T}$ be complete,

$Dom(\mathcal{T})$ will for each $u \in Sp(\mathcal{T})$ contain a set of representative continuations $u\alpha(\overline{d})$, where $\alpha(\overline{d})$ is taken from a *subset* of Σ_{Act} which in general depends on u. The ambition is that for each transition of the SUT, labeled $\alpha(\overline{p}), g$, from the state represented by u, the table contains at least one continuation $u\alpha(\overline{d})$ for a representative symbol $\alpha(\overline{d})$ with $\alpha(\overline{d}) \in [\![(\alpha(\overline{p}), g)]\!]$.

Second, in order to construct a parameterized system from an incomplete observation table, we present a technique to construct guards from representative symbols. This implies asking additional queries in order to determine guards as precisely as possible. Of course, we do not know *a priori* how many transitions leave a particular state, or how the guards partition symbols into equivalence classes. Therefore we start with a coarse default partitioning into equivalence classes, which is refined "by need". Whenever two words in the same equivalence class generate different reactions by the SUT, we split the equivalence class by introducing more guards.

In order to maintain a current hypothesis about guards, we augment the observation table $\mathcal{T}$ by a *labeling function* γ, which to each prefix $ua \in Dom(\mathcal{T})$ assigns a guarded action $\gamma_u(a)$. The idea is that the constructed parameterized system, after having processed the input word u, will process the input symbol a using a transition labeled by $\gamma_u(a)$. We make the natural requirements that $a \in [\![\gamma_u(a)]\!]$, and that the labeling function should suggest guards that make the resulting automaton completely specified and deterministic, i.e., for each $u \in Sp(\mathcal{T})$, we have

- $\bigcup_{ua \in Dom(\mathcal{T})} [\![\gamma_u(a)]\!] = \Sigma_{Act}$, and
- $ua, ua' \in Dom(\mathcal{T})$ implies either $[\![\gamma_u(a)]\!] = [\![\gamma_u(a')]\!]$ or $[\![\gamma_u(a)]\!] \cap [\![\gamma_u(a')]\!] = \emptyset$.

The addition of a labeling function makes it natural to strengthen the notion of consistency, to allow a unique parameterized system to be constructed from an observation table with a labeling function.

Definition 3. *A labeling function γ for an observation table $\mathcal{T}$ is* guard-consistent *if for any $ua, u'a' \in Dom(\mathcal{T})$ such that $u \approx_{\mathcal{T}} u'$ and $[\![\gamma_u(a)]\!] \cap [\![\gamma_{u'}(a')]\!] \neq \emptyset$, we have $ua \approx_{\mathcal{T}} u'a'$.*

Intuitively, whereas consistency states that extensions ua and $u'a$ in $Dom(\mathcal{T})$ of equivalent prefixes u and u' with the *same symbol* a should also be equivalent, guard-consistency requires that two symbols a, a', whose labeling functions *overlap* should have equivalent extensions in $Dom(\mathcal{T})$. Note that guard-consistency as a special case implies that $ua \approx_{\mathcal{T}} ua'$ whenever $ua, ua' \in Dom(\mathcal{T})$ and $[\![\gamma_u(a)]\!] = [\![\gamma_u(a')]\!]$.

We now have defined enough concepts to be able to define how to construct a parameterized system from an observation table with a labeling function.

Definition 4. *Let Act be a finite set of actions. Let $\mathcal{T}$ be a partitioned, closed, and consistent observation table, and let γ be a guard-consistent labeling function for $\mathcal{T}$. Define the parameterized system $\langle \mathcal{T}, \gamma \rangle / \approx_{\mathcal{T}}$ as $(Q, \longrightarrow, q_0, F)$, where*

- $Q = Dom(\mathcal{T})/\approx_{\mathcal{T}}$,
- $[u] \xrightarrow{\alpha(\overline{p}),g} [ua]$ *whenever* $ua \in Dom(\mathcal{T})$ *and* $\gamma_u(a) = (\alpha(\overline{p}), g)$, *and* u *is the principal prefix in* $[u]$,
- $q_0 = [\varepsilon]$, *and*
- $F = \{[u] \ : \ \mathcal{T}(u)(\varepsilon) = +\}$.

where for each equivalence class $[u]$ *we have designated a unique principal prefix* $u' \in [u]$ *with* $u' \in Sp(\mathcal{T})$. □

Note that guard-consistency guarantees that different choices of principal prefixes result in equivalent parameterized systems.

In general, there are many different guard-consistent labeling functions for a given observation table. We therefore define an additional criterion which constrains how conjuncts may occur in guards of a labeling function. In a table $\mathcal{T}$, define a *witnessing pair* for a prefix $u \in Sp(\mathcal{T})$, action α, and index i, to be a pair of prefixes $u\alpha(\overline{d}), u\alpha(\overline{d}') \in Dom(\mathcal{T})$ such that

- $u\alpha(\overline{d}) \not\approx_{\mathcal{T}} u\alpha(\overline{d}')$, and
- $\overline{d} = (d_1, \ldots, d_i, \ldots, d_n)$ and $\overline{d}' = (d_1, \ldots, d_i', \ldots, d_n)$ differ only in the ith parameter.

Definition 5. *A labeling function* γ *for* $\mathcal{T}$ *is* well-witnessed *if whenever* $\gamma_u(a) = (\alpha(\overline{p}), g)$ *then*

- *whenever* p_i *or* $\neg p_i$ *is a conjunct in* g, *then* $\mathcal{T}$ *contains a witnessing pair for* u, α, *and* i.
- *there is a conjunct* p_j *or* $\neg p_j$ *of* g *such that* $\mathcal{T}$ *contains a witnessing pair* $u\alpha(\overline{d}), u\alpha(\overline{d}')$ *for* u, α, *and* j, *such that* $\alpha(\overline{d}) \in [\![(\alpha(\overline{p}), g)]\!]$. □

Intuitively, the first requirement states that each conjunct of a guard g should be motivated by a witnessing pair in $\mathcal{T}$, which however need not contain a prefix that satisfies g. The second requirement states that g should be satisfied by the last symbol of at least one prefix in a witnessing pair.

We are now ready to state a theorem which relates a parameterized system $\langle \mathcal{T}, \gamma \rangle / \approx_{\mathcal{T}}$ constructed from an observation table $\mathcal{T}$, and the internal structure of the SUT.

We first adapt Theorem 1 to be sure that $\langle \mathcal{T}, \gamma \rangle / \approx_{\mathcal{T}}$ agrees with the observations.

Theorem 2. *Let* $\mathcal{T}$ *be a partitioned, closed, and consistent observation table, and let* γ *be an* $\approx_{\mathcal{T}}$*-compatible and guard-consistent labeling function for* $\mathcal{T}$. *If* $\mathcal{T}$ *is suffix-closed, then the parameterized system* $\langle \mathcal{T}, \gamma \rangle / \approx_{\mathcal{T}}$ *is conformant with* $\mathcal{T}$.

Proof. The theorem follows by adapting Theorem 1 to incomplete observation tables, and the requirement that $a \in [\![\gamma_u(a)]\!]$ for all $ua \in Dom(\mathcal{T})$. □

A more informative theorem, which can be seen as an analogue of Theorem 1 in [12], is as follows,

Theorem 3. *Let $\mathcal{T}$ be a partitioned, closed, consistent, and suffix-closed observation table, and let γ be a guard-consistent and well-witnessed labeling function for $\mathcal{T}$. Let $\langle \mathcal{T}, \gamma \rangle / \approx_{\mathcal{T}}$ be $(Q, \longrightarrow, q_0, F)$ with n states. Let $\mathcal{P} = (R, \longrightarrow', r_0, G)$ be any expanded parameterized system which is conformant with $\mathcal{T}$. Then $\mathcal{P}$ has at least n states and there is a surjective mapping h from R to Q such that*

- $h(r_0) = q_0$,
- $r \in G$ *iff* $h(r) \in F$,
- *if $\mathcal{P}$ has exactly n states then, whenever $h(r) = q$ and $q \xrightarrow{\alpha(\overline{p}),g} q'$, there are g', r' such that $r \xrightarrow{\alpha(\overline{p}),g'} r'$ with $h(r') = q'$ and $g' \Longrightarrow g$.*

This implies that if $\mathcal{P}$ has n states then $\langle \mathcal{T}, \gamma \rangle / \approx_{\mathcal{T}}$ has at most as many transitions as $\mathcal{P}$.

Optimization. The process of obtaining a well-witnessed labeling function may need a number of additional queries, which cause $Dom(\mathcal{T})$ to be extended. The requirement that $\approx_{\mathcal{T}}$ be an equivalence relation on $Dom(\mathcal{T})$ may then necessitate even more queries, which are not necessary for making γ well-witnessed. To allow to save queries, we allow prefixes in $Dom(\mathcal{T})$ to be classified as either *essential* or *auxiliary*. We now say that an observation table is *partitioned* if

- For each $ua \in Dom(\mathcal{T})$, there is an essential $ua' \in Dom(\mathcal{T})$ with $\gamma_u(a') = \gamma_u(a)$,
- ε is an essential prefix, and
- $\approx_{\mathcal{T}}$ is an equivalence relation on essential prefixes in $Dom(\mathcal{T})$.

4 An Algorithm for Inference of Parameterized Systems

In this section, we present an algorithm for inferring parameterized systems, based on the concepts introduced in Section 3.

The basic idea of our algorithm is to perform membership queries until we have a suffix-closed, partitioned, closed, and consistent observation table with a guard-consistent and well-witnessed labeling function. We can then construct a conjecture and pose an equivalence query. As long as the table does not satisfy some condition mentioned in Theorem 3, this is handled as follows.

- If $\mathcal{T}$ is not suffix-closed, i.e., there is a $(u, av) \in \mathit{Entries}(\mathcal{T})$ where $u \in Sp(\mathcal{T})$, such that $(ua, v) \notin \mathit{Entries}(\mathcal{T})$, then add (ua, v) to $\mathit{Entries}(\mathcal{T})$ (letting $\mathcal{T}(ua)(v) = \mathcal{T}(u)(av)$).
- If $\mathcal{T}$ is not partitioned, i.e., $\approx_{\mathcal{T}}$ is not an equivalence relation, then there are $u, u', u'' \in Dom(\mathcal{T})$ such that $u \approx_{\mathcal{T}} u'$, $u \approx_{\mathcal{T}} u''$ but $\mathcal{T}(u')(v) \neq \mathcal{T}(u'')(v)$ for some v. In this case, ask a membership query for uv, whose result is entered as $\mathcal{T}(u)(v)$ to determine whether u should be equivalent to u' or u''.
- If $\mathcal{T}$ is not closed, then for some $ua \in Dom(\mathcal{T})$ we have $ua \not\approx_{\mathcal{T}} u'$ for all $u' \in Sp(\mathcal{T})$. We then add ua to $Sp(\mathcal{T})$ by adding, for each $\alpha \in Act$, some word of form $ua\alpha(\overline{d})$ to $Dom(\mathcal{T})$, and let $\gamma_{ua}(\alpha(\overline{d})) = (\alpha, true)$. Priority

given to parameters $\overline{d}'$ for which $\alpha(\overline{d}')v \in \mathit{Dom}(\mathcal{T}(ua))$ for some v, since suffix-closedness then requires that $ua\alpha(\overline{d}') \in \mathit{Dom}(\mathcal{T})$.

– If $\mathcal{T}$ is not consistent, then we have two entries (ua, v) and $(u'a, v)$ in $\mathit{Entries}(\mathcal{T})$ with $\mathcal{T}(ua)(v) \neq \mathcal{T}(u'a)(v)$ but $u \approx_{\mathcal{T}} u'$. Then add (u, av) and (u', av) to $\mathit{Entries}(\mathcal{T})$ and enter the results from $\mathcal{T}(ua)(v)$ and $\mathcal{T}(u'a)(v)$, respectively.

The table must also be equipped with a labeling function γ, which is maintained during the algorithm. Initially, for each $u \in \mathit{Sp}(\mathcal{T})$ and each action α, we choose some values $\overline{d}$ for the parameters of α, and let $u\alpha(\overline{d}) \in \mathit{Dom}(\mathcal{T})$ with $\gamma_u(\alpha(\overline{d})) = (\alpha, \mathit{true})$. Whenever we add a prefix $u\alpha(\overline{d})$ to $\mathit{Dom}(\mathcal{T})$ the labeling function is updated in one of two ways. If there is not yet a prefix $u\alpha(\overline{d}') \in \mathit{Dom}(\mathcal{T})$ for any $\overline{d}'$ we let $\gamma_u(\alpha(\overline{d})) = (\alpha, \mathit{true})$, otherwise we let $\gamma_u(\alpha(\overline{d})) = \gamma_u(\alpha(\overline{d}'))$, where $\alpha(\overline{d}')$ is the existing symbol such that $\alpha(\overline{d}) \in \left[\!\left[\gamma_u(\alpha(\overline{d}'))\right]\!\right]$.

If $\mathit{Dom}(\mathcal{T})$ contains only one prefix $u\alpha(\overline{d})$ for each u and α, then γ is well-witnessed. However, if another prefix $u\alpha(\overline{d}')$ is entered, for which $u\alpha(\overline{d}) \not\approx_{\mathcal{T}} u\alpha(\overline{d}')$, this destroys the guard-consistency. We then have to refine the labeling function γ, and possibly also the partitioning into equivalence classes.

If γ is not guard-consistent, this may be because there are u, a, and a' such that $\gamma_u(a) = \gamma_u(a')$ but $ua \not\approx_{\mathcal{T}} ua'$. Let $\gamma_u(a)$ be $(\alpha(\overline{p}), g)$. In this case, we must split the guard g so that a and a' are assigned disjoint guards. In order to find an appropriate parameter for the splitting, and to keep γ well-witnessed, we find (e.g., by binary search) two tuples, $\overline{d} = (d_1, \ldots, 1, \ldots, d_n)$ and $\overline{d}' = (d_1, \ldots, 0, \ldots, d_n)$, of parameter values of α, with $\alpha(\overline{d}), \alpha(\overline{d}') \in [\![\gamma_u(a)]\!]$, which differ only in some parameter (with index, say, i), such that $\mathcal{T}(u\alpha(\overline{d}))(v) \neq \mathcal{T}(u\alpha(\overline{d}'))(v)$ for some v. We then add $(u\alpha(\overline{d}), v)$ and $(u\alpha(\overline{d}'), v)$ to $\mathit{Entries}(\mathcal{T})$, and update the labeling function so that all $ua'' \in [\![\gamma_u(a)]\!]$ now labeled by the guard $g \wedge p_i$ or $g \wedge \neg p_i$.

A second source of guard-inconsistency is that we can have two equivalent prefixes in $\mathit{Sp}(\mathcal{T})$ which have different partitionings of the next symbols, induced by the labeling function. It must then always be the case that there exist u, u', a, and a' such that $ua, u'a' \in \mathit{Dom}(\mathcal{T})$, $u \approx_{\mathcal{T}} u'$, and $a' \in [\![\gamma_u(a)]\!]$ but $\mathcal{T}(ua)(v) \neq \mathcal{T}(u'a')(v)$ for some v. A membership query for $ua'v$ should clarify the situation, either giving rise to a guard-inconsistency, or causing $u \not\approx_{\mathcal{T}} u'$ (and continuing processing it as an inconsistency).

When we have a partitioned, suffix-closed, consistent, and closed table with a well-witnessed and guard-consistent labeling function, we can construct a conjecture as described in Definition 4. The conjecture is provided to the oracle in an equivalence query and the oracle in turn either gives an affirmative answer or a counter-example. In the first case, the algorithm terminates and outputs the correct model. In the second case, the oracle returns a counter-example, i.e., a word u such that $\mathit{Obs}(u) = +$ but the provided automaton does not accept u (or vice versa). As in the standard algorithm of Angluin, we enter all prefixes of

u into $Dom(\mathcal{T})$. This will subsequently cause either an inconsistency and hence a "new" state, or a guard-inconsistency and hence a "new" transition.

Algorithm Query complexity. We estimate the complexity of our algorithm in terms of a minimal expanded parameterized system which accepts exactly the language as the SUT. Let n be its number of states, let m be the number of transitions, let $|Act|$ be the number of actions in Act, let $|\Sigma_{Act}|$ be the number of symbols in Σ_{Act}, and let c be the length of the longest counter-example received from the oracle.

The expected bottle-neck in practice for an inference algorithm is the number of membership and equivalence queries, since queries often involve comparatively slow communication with an external device. Let us first estimate the number of equivalence queries. An equivalence query can either give rise to

- an inconsistency which results in a new state; this can occur at most n times, or
- a guard-inconsistency which results in splitting a guard; this can occur at most $m - n|Act|$ times.

Hence the algorithm performs at most $n + m - n|Act|$ equivalence queries.

Let us then estimate the number of membership queries. The number of membership queries required are dependent on the number of prefixes in $Dom(\mathcal{T})$ and the maximum number of suffixes in any $Dom(\mathcal{T}(u))$. Each $Dom(\mathcal{T}(u))$ contains at most n suffixes, since each time we add a new suffix to $Dom(\mathcal{T}(u))$ we separate at least a pair of prefixes into different equivalence classes. The number of prefixes in $Dom(\mathcal{T})$ is at most

- one for each equivalence class; totally n, plus
- one for each state and action, plus an extra essential pair of prefixes as witness for each transition, in total $n|Act| + 2m$, plus
- prefixes of counterexamples, in total $c(n + m - n|Act|)$.

Hence the number of membership queries performed by the algorithm is $O(cmn)$ (since $n|Act| \leq m$). We can contrast this with a naive application of Angluins algorithm, which in the worst case requires $O(cn^2|\Sigma_{Act}|)$ membership queries. Thus, whereas a naive use of Angluins algorithm uses a number of membership queries which grows linearly with $|\Sigma_{Act}|$, i.e., exponentially in the arity of actions, our algorithm grows exponentially only with the number of parameters of an action that is used in guards of transitions. It should be remarked that Angluin's treatment of counterexamples is poorly optimized, resulting in the factor c in the worst-case bound. Rivest and Shapire [17] have presented techniques for replacing the factor c by $\log c$, which should apply also to our algorithm.

5 Experimental Results

We are interested in examining how the performance of the inference algorithm for parameterized systems depends on the number of parameters that occur

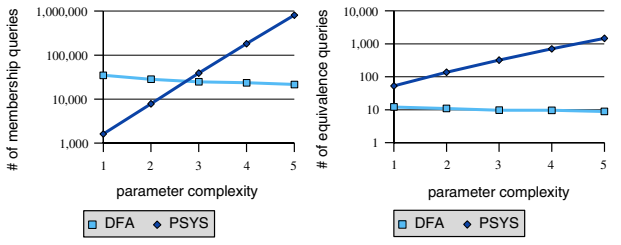


Fig. 2. Experimental results on random generated parameterized systems with 50 states and 5 parameters

in guards in the transitions of the system and how it compares with a naive application of Angluin's algorithm. Let us first define a measure for this. Let the *parameter complexity* for a state q and action α of a parameterized system, be the total number of different parameters used in guards on transitions from q labeled $\alpha(\overline{p})$. We want to investigate how the parameter complexity effects the number of membership and equivalence queries required by the algorithm. For this purpose, we have implemented our inference algorithm for parameterized systems. The implementation is in C++ as an extension of the LearnLib tool [19], developed at Dortmund University.

We measure performance on randomly generated parameterized systems and a small model of an instance of a protocol provided by Mobile Arts AB (see Fig. 1). The protocol was first modeled in LOTOS and then transformed into a DFA by the CAESAR/ALDEBARAN Development Package [22]. The protocol is a small fragment of the Network Presence Center (NPC) product of the company. The NPC is a middle-ware product to allow Mobile Network Operators to provide various presence information from the GSM network. The parameterized system model of the protocol has 4 states, one action with arity 12 and another with arity 7. The first action has on average parameter complexity 0 and the second 0.5. In the randomly generated systems we have used actions with arity 5, and generated automata in which each state-action pair has the same parameter complexity. We have varied the parameter complexity between 1 and 5. The systems has then been inferred both by our algorithm and by Angluin's algorithm. The results of the experiments are summarized in Figure 2, where the left diagram shows the number of membership queries, and the right diagram shows the number of equivalence queries.

The left diagram shows that the number of membership queries for our algorithm grows exponentially with the parameter complexity of the system, whereas it is independent of parameter complexity for Angluin's original algorithm. For a parameter complexity of less than 3, our algorithm performs better, but when parameter complexity increases, the overhead of our algorithm makes it clearly worse than Angluin's. The right diagram shows that our algorithm always performs more equivalence queries than Angluin's.

Applying Angluin's algorithm to Mobile Arts' protocol fragment gives rise to 76000 membership queries and 3 equivalence queries, while our algorithm only

requires 21 membership queries and 4 equivalence queries. The reason for this difference is the relatively low parameter complexity in the overall system in comparison to the high arity of its actions.

The higher number of equivalence queries for our algorithm is an expected consequence of the observation that our algorithm allows to construct equivalence queries that are based on less complete information than Angluin's algorithm. In particular, we allow equivalence queries even if the refinement of equivalence classes of symbols is not completed. For higher parameter complexity (4 or 5), the difference in number of equivalence queries is significant. We believe that this explains the sharp growth of membership queries for parameter complexities 4 and 5, since a large number of equivalence queries gives rise to an explosion in membership queries that are caused by prefixes of counterexamples.

6 Conclusions

In this paper, we have adapted techniques for inference of finite automata from sets of observations, in order that they perform better for state machines whose symbols are generated from a small set of actions, each of which has a set of parameters. Our algorithm tries to find representative observations, from which we infer guards of transitions by techniques for inferring boolean expressions. Thus, our work indicates a way to combine techniques for inferring properties of data types with regular inference techniques for inferring reactive behavior. Our algorithm requires less observations in the case that only a subset of parameters are used to determine the behavior of the machine at each transition. Future work includes to improve the handling of counterexamples in our tool, and to evaluate our techniques on a realistic communication protocol module.

Our framework limits us to handling inputs but not outputs. We therefore suggest possible solutions to include these. One approach is to infer a Mealy machine like Steffen et al. [23] but with our framework of handling parameterized input actions. The other approach is to use our framework but encode the input and output into parameterized actions of a parameterized system. This will of course blow up the alphabet, but the dependences between input and output will be recorded in boolean formulas which may lead to very compact models.

Acknowledgement. At last we would like to thank Bernhard Steffen for helpful hints and discussions.

References

1. Broy, M., Jonsson, B., Katoen, J.P., Leucker, M., Pretschner, A., eds.: Model-Based Testing of Reactive Systems. Volume 3472 of LNCS. Springer Verlag (2004)
2. Ball, T., Rajamani, S.: The SLAM project: Debugging system software via static analysis. In: Proc. 29^{th} ACM Symp. on Principles of Programming Languages. (2002) 1–3

3. Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Pasareanu, C., Robby, Zheng, H.: Bandera: Extracting finite-state models from java source code. In: Proc. 22nd Int. Conf. on Software Engineering. (2000)
4. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. 29^{th} ACM Symp. on Principles of Programming Languages. (2002) 58–70
5. Holzmann, G.: Logic verification of ANSI-C code with SPIN. In: SPIN Model Checking and Software Verification: Proc. 7^{th} Int. SPIN Workshop. Volume 1885 of LNCS., Springer Verlag (2000) 131–147
6. Hagerer, A., Hungar, H., Niese, O., Steffen, B.: Model generation by moderated regular extrapolation. In: Proc. FASE '02. Volume 2306 of LNCS., Springer Verlag (2002) 80–95
7. Hungar, H., Niese, O., Steffen, B.: Domain-specific optimization in automata learning. In: Proc. 15^{th} Int. Conf. on Computer Aided Verification. (2003)
8. Groce, A., Peled, D., Yannakakis, M.: Adaptive model checking. In: Proc. TACAS '02. Volume 2280 of LNCS., Springer Verlag (2002) 357–370
9. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. In: FORTE/PSTV, Kluwer (1999) 225–240
10. Ammons, G., Bodik, R., Larus, J.: Mining specificatoins. In: Proc. 29^{th} ACM Symp. on Principles of Programming Languages. (2002) 4–16
11. Cobleigh, J., Giannakopoulou, D., Pasareanu, C.: Learning assumptions for compositional verification. In: Proc. TACAS '03. Volume 2619 of LNCS., Springer Verlag (2003) 331–346
12. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75** (1987) 87–106
13. Balcázar, J., Daz, J., Gavaldá, R.: Algorithms for learning finite automata from queries: A unified view. In: Advances in Algorithms, Languages, and Complexity. Kluwer (1997) 53–72
14. Dupont, P.: Incremental regular inference. In: ICGI. Volume 1147 of LNCS., Springer (1996) 222–237
15. Gold, E.M.: Language identification in the limit. Information and Control **10** (1967) 447–474
16. Kearns, M., Vazirani, U.: An Introduction to Computational Learning Theory. MIT Press (1994)
17. Rivest, R., Schapire, R.: Inference of finite automata using homing sequences. Information and Computation **103** (1993) 299–347
18. Trakhtenbrot, B., Barzdin, J.: Finite automata: behaviour and synthesis. North-Holland (1973)
19. Raffelt, H., Steffen, B., Berg, T.: Learnlib: a library for automata learning and experimentation. In: FMICS '05, ACM Press (2005) 62–71
20. Gold, E.M.: Complexity of automaton identification from given data. Information and Control **37** (1978) 302–320
21. Blom, J., Jonsson, B.: Automated test generation for industrial erlang applications. In: Proc. 2003 ACM SIGPLAN workshop on Erlang, Uppsala, Sweden (2003) 8–14
22. Garavel, H., Lang, F., Mateescu, R.: An overview of cadp 2001 (2002) Newsletter.
23. Steffen, B., Margaria, T., Raffelt, H., Niese, O.: Efficient test-based model generation of legacy systems. In: HLDVT'04, IEEE Computer Society Press (2004) 95–100

Automated Support for Building Behavioral Models of Event-Driven Systems

Benet Devereux and Marsha Chechik

Department of Computer Science, University of Toronto,
Toronto, Ontario, Canada
{benet, chechik}@cs.toronto.edu

Abstract. Programmers understand a piece of software by building simplified mental models of it. Aspects of these models lend themselves naturally to formalization – e.g., structural relationships can be partly captured by module dependency graphs. Automated support for generating and analyzing such structural models has proven useful. For event-driven systems, behavioral models, which capture temporal and causal relationships between events, are important and deserve similar methodological and tool support. In this paper, we describe such a technique. Our method supports building and elaboration of behavioral models, as well as maintaining such models as systems evolve. The method is based on model-checking and witness generation, using strategies to create goal-driven simulation traces. We illustrate it on a two-lift/three-floor elevator system, and describe our tool, Sawblade, which provides automated support for the method.

1 Introduction

Programs larger than a few tens of lines are generally far too complex to be understood in full by a single person. In place of complete understanding, programmers use simplified mental models [18] – a representation of some part of the program's structure or function at a high enough level of abstraction to be readily understood. One heuristic is that they should be small enough to fit on a whiteboard [25]. Mental models are informal and cannot always be completely expressed by a formal structure; however, it has often been found useful to create formal structures based upon programmers' mental models and to use them to aid construction and understanding of code. Some examples of mental models used by programmers, and their corresponding formal artifacts, are discussed below.

Modules and dependencies. Decomposition into modules that communicate via interfaces is standard software engineering practice. Module A depends on module B if any of the functions in A use a function or data defined in B. Considerable research has gone into automatically extracting a graph recording all dependencies between modules from the source code [8, 23, 24], helping programmers navigate this graph [13] or select fragments that are most relevant to a particular aspect [28].

Design patterns and architectural patterns. Design patterns describe and catalogue common relationships between groups of objects in object-oriented programs [14]. Design patterns have a formal representation as fragments of object modelling graphs;

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 122–138, 2006.

formalizing the observations has led to faster program comprehension and better communication between programmers. Work on extracting design patters from source code has also been done [21]. *Architectural patterns* relate components-and-connectors type diagrams to standard styles of decomposition, such as layers or pipes-and-filters. There is research on automated exploration of architecture and automatic comparison with a conceptual pattern [13, 25].

The models described above are mostly *structural*. However, *behavior* is as important a part of understanding a program as structure, particularly for reactive concurrent systems. In this paper, we address the problem of providing tool support for constructing behavioral models of such complex systems.

Global propositions about a system's behavior can be expressed using temporal logic [22] and automatically verified using model-checking [9]. The strength of this method is that it can be used both for assertions about *some* behavior and about *all possible* behaviors. It has two weaknesses: expressing properties can become difficult, though property patterns [12] help tame this difficulty; and, more importantly, even using property patterns, it is very hard to *guess* which properties might be both valid and useful for understanding.

Query-checking [5] is a technique for searching for interesting temporal logic properties. It enables answering user questions such as "What property P is true everywhere?" or "If event X happens, what property Q eventually becomes true sometime after X?" However, query-checking requires considerable intervention and technical knowledge. Furthermore, its output can be a large propositional formula, which is hard to interpret intuitively.

A potential candidate for behavioural models is *scenarios* [20]. Scenarios have been shown to be useful for expressing requirements and for communicating between stakeholders [16, 19]. Scenarios can capture not only sequences of events that the system allows, but also those that it prohibits, exact causal relationship between events, etc. Running the program – with the aid of a simulator or test-driver to provide inputs – generates a large number of scenarios which are certainly true, but do not yield the kind of simple, general knowledge about the system's behavior we would like in building a mental model. They lack any notion of causation between events, or of necessity or impossibility of sequences of events. These richer concepts are essential for behavioral models that support understanding and evolution of the system.

Our goal is to bridge this gap: to provide a methodology and tool for finding and validating rich scenarios that describe not just sequences of events but causal relationships between them. We need to be able to vary the level of *granularity* of scenarios – what events we distinguish – and also the *scope*, to ignore actions of parts of the system that are not considered relevant.

Furthermore, we want to go beyond simply finding and validating such scenarios: our methodology aims to help the user in elaborating scenarios by finding others which are stronger, or more detailed. Finally, since a major use of mental models is during software evolution, our methodology must also help change these scenarios along with evolving systems.

Contributions. In this paper, we propose a methodology and tool based on temporal logic, model-checking, and witness generation. Our techniques do not work directly on

the program code; instead, we assume that a finite-state model of the program has been constructed, e.g., using the techniques of [1, 11], and that this model is small enough to be analyzable by existing model-checkers.

We illustrate our methodology using a two-lift/three-floor elevator system [2]. Rather than requiring direct use of temporal logic, our method uses a simple language of events and causal relationships between them. This language itself is not new: its features are drawn from property patterns and use-case maps [4]. The modeling language can be used to express scenarios, and a translation into temporal logic is used for automatic validation by a model-checker. Once scenarios have been found and validated, they can be elaborated. We describe a set of patterns for moving from a validated scenario to stronger and richer scenarios. Application of these patterns relies on generating the most useful traces of the program that help the user guess more elaborate scenarios. The notion of a useful trace is user-specified, and this specification is used by the witness-generation component of the model-checker to carry out a strategy-directed search for interesting traces.

Maintaining scenarios across change is done by representing change as an annotation of the new system, indicating how its state transitions have changed from the old. Once this is done, useful traces – in this case, those that highlight most effectively and minimally the differences in behavior, where they exist – can be searched for by the model-checker using strategies as well.

Support for this method is provided by our tool Sawblade, built on top of a model-checker XChek [6].

Structure. The rest of this paper is organized as follows: in Section 2, we give background material on temporal logic and model-checking. In Section 3, we present a language for scenario-like behavioural models and its translation into temporal logic. We also discuss the elevator system which is the running example in this paper. In Section 4, we describe witness generation and strategies for helping produce the "most interesting" witnesses. In Section 5, we describe the methodology for elaborating scenarios. In Section 6, we give our formal definition of the annotation of a changed system, and in Section 7, describe the methodology for transforming scenarios across change. We describe Sawblade, a tool supporting this methodology, in Section 8 and conclude the paper in Section 9.

2 Background

In this section, we review the basics of temporal logic model-checking, presenting the semantics of the temporal logic CTL and the definition of witnesses for existential CTL properties.

Analysis of data-driven, run-to-completion programs is predicative, examining the relation between the program's input and output. Analysis of a reactive program, however, must examine the infinite behaviours of the program, and how its behaviours are affected by input from its environment. Temporal logic is helpful for intuitively expressing properties of infinite behaviours, and, for finite-state models, *model-checking* provides a useful tool for automatically deciding satisfaction of temporal logic properties by those models.

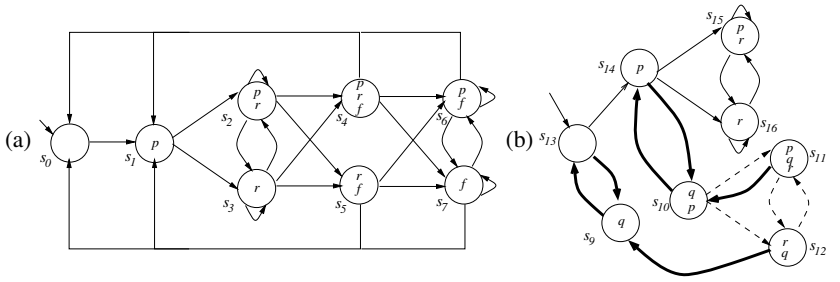


Fig. 1. (a) A Kripke structure; (b) A Kripke structure with `diff`, based on a fragment of the model in (a)

Kripke Structures and CTL. A *Kripke structure* is an abstract model of a reactive system. Formally, it is a tuple (S, s_0, R, I, V) where S is a (finite) set of states; s_0 is the initial state; $R \subseteq S \times S$ is the transition relation; V is a set of atomic propositions; $I : S \rightarrow 2^V$ is a labeling function that associates each state with the atomic propositions true in that state.

An example Kripke structure is shown in Figure 1(a). In this model, $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$, $V = \{p, r, f\}$, and $(s_0, s_1) \in R$. Atomic propositions not shown in a state are assumed to be false, e.g., p, r and f are false in s_0.

For each $s \in S$, the transition relation defines a *successor set* $Img(s) = \{t \mid R(s,t)\}$ of states reachable in one step from s; the *predecessor set* is $Img^{-1}(s) = \{t \mid R(t,s)\}$. A *path* p is an infinite sequence $p_0 p_1 p_2 \ldots$ of states. The set of paths $\mathcal{P}(s)$ of a state s in some Kripke structure M contains all the infinite sequences of states possible in M: $p \in \mathcal{P}(s) \Leftrightarrow p_0 = s \wedge \forall i \in \mathbb{N} \cdot p_{i+1} \in Img(p_i)$.

Computation Tree Logic (CTL) [10] is a temporal logic used to state properties of the (infinite) paths of Kripke structures. The set of CTL formulas over a set of atomic propositions (variables) V consists of the sentences defined by the following grammar:

$$C ::= p \in V \mid \neg C \mid C \wedge C \mid C \vee C \mid \mathbf{EX}\ C \mid \mathbf{AX}\ C \mid \mathbf{E}[C\ \mathbf{U}\ C] \mid \mathbf{A}[C\ \mathbf{U}\ C] \mid \mathbf{EF}\ C \mid \mathbf{AF}\ C \mid \mathbf{EG}\ C \mid \mathbf{AG}\ C$$

The symbols $\mathbf{AX}, \mathbf{EG}$, etc., are called temporal operators. The $\mathbf{A}$ or $\mathbf{E}$ indicates whether the following symbol is to be interpreted over all future paths, or some future paths; $\mathbf{X}$ stands for "next", $\mathbf{F}$ for "future", $\mathbf{U}$ for "until", $\mathbf{G}$ for globally; thus $\mathbf{AX}\varphi$ means "in all next states, φ holds", and $\mathbf{EF}\psi$ means "there is a future path along which, at some point, ψ holds".

The CTL satisfaction relation $\models$ is defined between states of a Kripke structure and CTL formulas. Its definition is given in Figure 2(a). Note that only $\mathbf{EX}$, $\mathbf{EU}$ and $\mathbf{EG}$ are presented. Others can be derived from these via simple identities [9]. For instance, $\mathbf{EF}\varphi \Leftrightarrow \mathbf{E}[true\ \mathbf{U}\ \varphi]$. Also note the operator $\mathbf{EU}_i$; informally, $\mathbf{E}[\varphi\ \mathbf{U}_i\ \psi]$ means that there exists a path along which ψ becomes true *no later than* at step i, and until that point, φ holds.

For instance, in the structure of Figure 1(a), we can ask whether it is possible to reach a state where f holds: $s_0 \models \mathbf{EF}\ f$. One such state is s_5, so the property holds.

Figure 2(b) shows some useful CTL identities which will be used later on. They are straightforward consequences of the semantics. Note that $\mathbf{EF}_i\varphi \Leftrightarrow \mathbf{E}[true\ \mathbf{U}_i\ \varphi]$.

(a)

$$
\begin{aligned}
s \models p &\Leftrightarrow p \in I(s) \\
s \models \neg\varphi &\Leftrightarrow s \not\models \varphi \\
s \models \varphi \wedge \psi &\Leftrightarrow s \models \varphi \wedge s \models \psi \\
s \models \mathbf{EX}\varphi &\Leftrightarrow \exists p \in \mathcal{P}(s) \cdot p_1 \models \varphi \\
s \models \mathbf{EG}\varphi &\Leftrightarrow \exists p \in \mathcal{P}(s) \cdot \forall i \cdot p_i \models \varphi \\
s \models \mathbf{E}[\varphi \mathbf{U} \psi] &\Leftrightarrow \exists p \in \mathcal{P}(s) \cdot \exists i \cdot (p_i \models \psi) \wedge \\
&\qquad \forall j < i \cdot p_j \models \varphi \\
s \models \mathbf{E}[\varphi \mathbf{U}_i \psi] &\Leftrightarrow \exists p \in \mathcal{P}(s) \cdot \exists j \leq i \cdot (p_j \models \psi) \wedge \\
&\qquad \forall k < j \cdot p_k \models \varphi
\end{aligned}
$$

(b)

$$
\begin{aligned}
&\mathbf{EF}\varphi \Leftrightarrow \varphi \vee \mathbf{EX}\ \mathbf{EF}\varphi \\
&\mathbf{EF}_0\varphi \Leftrightarrow \varphi \\
&\mathbf{EF}_i\varphi \Leftrightarrow \mathbf{EX}\ \mathbf{EF}_{i-1}\varphi \ \text{ if } i > 0 \\
&\mathbf{EF}_i\varphi \Rightarrow \mathbf{EF}\varphi \ \text{ for all } i \\
&\mathbf{EG}\ \varphi \Leftrightarrow \varphi \wedge \mathbf{EX}\ \mathbf{EG}\ \varphi
\end{aligned}
$$

Fig. 2. (a) Semantics of CTL. (b) Useful CTL identities.

The set of all states in a model M which satisfy a given property φ is denoted by $[\![\varphi]\!]^M$, or just $[\![\varphi]\!]$ when the model is implicit. The semantics of CTL can be expressed entirely in terms of $[\![\cdot]\!]$ rather than quantification over paths. For instance, $[\![\mathbf{EX}\varphi]\!] = \{Img^{-1}(s) \mid s \in [\![\varphi]\!]\}$, and $\mathbf{EF}\varphi$ is the least fixed-point of Img^{-1} applied to $[\![\varphi]\!]$.

Witnesses to CTL Properties. In first-order logic, an existential assertion $\exists x \cdot Q(x)$ can be proven by exhibiting a *witness* – an element in the domain of the predicate Q which makes Q true. Since CTL properties are expressed in a fragment of first-order logic, this proof method can be applied to them as well. For example, a witness for the property $\mathbf{EF}f$ in the model of Figure 1(a) is s_0, s_1, s_3, s_5.

Any CTL property whose semantics is entirely expressible using existential quantifiers where the negation is pushed to the level of atomic propositions, can be proven by exhibiting an instantiation for all the existential quantifiers over paths. Though CTL semantics is expressed over infinite paths, a witness is always made up of finite paths or finite prefixes followed by finite repeating suffixes [9]. For example, the witness for $s_0 \models \mathbf{EX}p$ for the model in Figure 1(a) is a two-step path, where the second step is a successor t such that $(s_0, t) \in R$ (state s_1 in our example). The witness for $s_1 \models \mathbf{EG}p$ is infinite, and, in the case of the model in Figure 1(a), consists of a loop $s_1, s_2, s_4, s_1, \ldots$. In the rest of the paper, we use "witness" to refer to *either* the necessary finite segments or to infinite paths that begin with such segments; the correct interpretation will be clear from the context. Also, we only consider properties *linear witnesses* [3], i.e., a single path through the states of the model that suffices as a proof. This restriction is for the sake of simplicity of presentation, and is not a constraint on the method discussed; our results can also be extended to witnesses with branching structure [15].

Determining whether a CTL formula has a linear witness is NP-hard [3]; a sublanguage of CTL which always has linear witnesses is given by the following grammar:

$$
\begin{aligned}
A &::= p \in V \mid \neg A \mid A \wedge A \mid A \vee A \\
T &::= A \mid \mathbf{EX}T \mid \mathbf{EF}T \mid \mathbf{E}[A\ \mathbf{U}\ T] \mid T \vee T
\end{aligned}
$$

For example, the witness to $\mathbf{E}[r\ \mathbf{U}\ \mathbf{EX}p]$ in state s_0 of the model in Figure 1(a) is linear, whereas the witness to $\mathbf{EX}p \wedge \mathbf{EX}\neg p$ in state s_1 is not.

A *counterexample* is a witness to the *negation* of a property. Let φ be a formula with a witness, and let $\psi = \neg\varphi$. If ψ does not hold in some state s, then a counterexample to ψ can be computed; further, if φ has a linear witness, then ψ has a linear counterexample.

3 Scenario Language

In this section, we describe the syntax and semantics of a simple language of scenarios. The language allows expression of causal relationships between events, under qualifying conditions. Its semantics is a translation into CTL, so that scenarios can be automatically validated.

To illustrate the concepts in this paper, we use a simple two-lift, three-floor elevator system with a central controller. Each of the elevators, E_1 and E_2, can be standing still, or moving up or down; its door can be open or closed. It has a record of the floors it is still obliged to visit – an elevator E_i must visit a floor if either (1) its internal button for that floor was pressed, or (2) the controller received a call from a landing-button on that floor and assigned E_i to service it. The controller assigns calls to elevators based on a heuristic estimating which will arrive first.

3.1 Syntax

The basic entities of mental models of behaviour are *conditions* – the state of a program spanning some nonzero number of steps in time – and *events* – changes between one state and the next [17]. Some of the events and conditions of the elevator system are shown in Figure 3(a). Note that events do not have to be independent of each other, e.g., *floor*=2 $\Leftrightarrow$ (E_1.*floor*=2 $\vee$ E_2.*floor*=2).

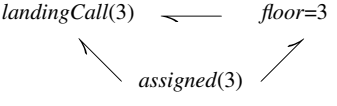
The fundamental relationships between events are temporal and causal. We consider the following to be the atomic relationships between events A and B:

$A \rightsquigarrow B$ A and B can happen, and B can follow A.
$A \rightharpoonup B$ A and B can happen, and if A happens, then B must happen sometime in the future.
$A \leftharpoondown B$ A and B can happen, and if B happens, then A must have happened prior to B.

Also, $A \rightleftharpoons B$ means that both $A \rightharpoonup B$ and $A \leftharpoondown B$. Composition of relationships is transitive: writing $A \rightharpoonup B \rightharpoonup C$ means that $A \rightharpoonup B$ and $B \rightharpoonup C$. For example, the graphical expression shown in Figure 3(b) denotes (*landingCall(3)* $\leftharpoondown$ *floor*=3) $\wedge$ (*landingCall*(3) $\leftharpoondown$ *assigned*(3)) $\wedge$ (*assigned*(3) $\rightharpoonup$ *floor*=3). That is, an elevator arrives at floor 3 only because of a call to floor 3. Also, floor 3 is assigned to an elevator only because of a call; and assignment of a floor always causes its service by an elevator.

Events	
init	the elevator is started up
landingCall(3)	there is a call for an elevator on floor 3
liftCall(3)	there is a call inside an elevator for floor 3
E_1.*floor* = 1	elevator 1 arrives on floor 1
floor= 2	*either* elevator arrives on floor 2
assigned(3)	the controller assigns a call to one of the elevators.
Conditions	
E_1.*up*	elevator 1 is moving up
outstandingCall(3)	there is an unserviced call for floor 3

(a)



(b)

Fig. 3. (a) Some elevator events and conditions; (b) An example graphical expression

Causal relationships can be *absolute* or *conditional*: either $A \rightharpoonup B$ in any case, or $A \rightharpoonup B$ while a condition c is satisfied, that is, if A happens while condition c is true, then *either* B eventually happens *or*, before that, c becomes false. We denote this by $c[A \rightharpoonup B]$. This situation is called an *exception*. In addition, we want to allow representation of exceptions which are due to *events*. If A leads to B unless event C happens, we write $(A \rightharpoonup B) \downarrow C$. We can further generalize this by defining *scopes* as in the temporal logic patterns framework [12, 27], e.g., $A \rightharpoonup B$ between (conditions or events) P and Q.

3.2 Translation

Since Kripke structures deal only with propositions, and not events, we must explicitly encode events as changes of state. For instance, in the elevator, *floor*=1 is a state variable: the elevator *arrives* at floor 1 when this variable *becomes* true. We assume that the structure, where needed, is annotated with event variables, which become true for a single time-step whenever the event occurs, and false once it ceases to occur.

Since any scenario expression can be represented by a conjunction of atomic binary expressions, we only describe the translation of atomic expressions into CTL.

If $B \rightsquigarrow C$, then there are three properties to be checked: (1) B can occur; (2) C can occur; (3) C can follow B. These are subsumed by determining whether there exists some path from the initial state along which B occurs at some point; and whether, once B occurs, C may occur. Formally, $B \rightsquigarrow C = \mathbf{EF}(B \wedge \mathbf{EF}\ C)$. For instance, we can check *init* $\rightsquigarrow$ (*floor*=3) by asking the model-checker whether $\mathbf{EF}(\mathit{init} \wedge \mathbf{EF}$ *floor*=3) holds. Since *init* is necessarily true of s_0, this reduces to $\mathbf{EF}$ *floor*=3.

The translation of the remaining constructs into CTL is shown in Table 1. **W/C** indicates whether the translation has a linear witness (W), a linear counterexample (C), or neither (–). If $B \rightharpoonup C$, then not only can both B and C occur, but when B occurs, then C *must* occur at some point after it. As an example, we can ask whether *call*=3 $\rightharpoonup$ *floor*=3 is a valid scenario; the model-checker determines whether for any state where *call*=3 occurs, each future path eventually reaches a state where *floor*=3. If $B \leftharpoonup C$, then either C never occurs, or on any path where C does occur, between the initial state and the first occurrence of C, and between any two occurrences of C, there is an occurrence of B. $c[B \rightharpoonup D]$ means that if B occurs *while* c is true, then along all future paths, c holds until either D occurs, or c becomes false (the exception condition). For $(B \rightharpoonup C) \downarrow E$, B must either lead to an occurrence of C or an occurrence of E; C must be possible, but E need not be.

Table 1. CTL semantics of atomic scenarios

Scenario	CTL translation	W/C
$B \rightsquigarrow C$	$\mathbf{EF}(B \wedge \mathbf{EF}\ C)$	W
$B \rightharpoonup C$	$\mathbf{AG}(B \Rightarrow \mathbf{AF}\ C)$	C
$B \leftharpoonup C$	$\mathbf{AG}(\mathit{init} \vee C \Rightarrow (\mathbf{AG}\neg C \vee \mathbf{A}[\neg C\ \mathbf{U}\ B]))$	–
$c[B \rightharpoonup D]$	$\mathbf{AG}(B \wedge c \Rightarrow \mathbf{A}[c\ \mathbf{U}\ D \vee \neg c])$	C
$(B \rightharpoonup C) \downarrow E$	$\mathbf{AG}(B \wedge \neg E \Rightarrow \mathbf{AF}(C \vee E))$	C

Once relationships between events are discovered, they can be immediately validated by the model-checker. This leaves open, however, the question of how to find such relations and how to make them more precise – elaborate them. We address this issue in Section 5.

4 Witnesses and Strategies

In this section, we discuss how to define strategies for constructing "interesting" witnesses. We also discuss optimality of a witness-generation strategy with respect to objectives which may not be expressible as part of CTL.

4.1 Witness Generation

In Section 2, we defined witnesses. We now discuss their effective computation.

We start by defining *annotated witnesses*. An annotated witness is a sequence π of pairs $(\pi_0, \Phi_0), (\pi_1, \Phi_1), ...$ where π_i is a state and Φ_i a set of CTL formulas. The formulas Φ_i are *proof-obligations* – informally, properties which, at step π_i, still need to be demonstrated by the witness. For example, the annotated witness to $s_0 \models \mathbf{EF}\ f$ for the model in Figure 1(a) is

$$(s_0, \{\mathbf{EF}\ f\}) \rightarrow (s_1, \{\mathbf{EF}_2\ f\}) \rightarrow (s_2, \{\mathbf{EF}_1\ f\}) \rightarrow (s_5, \{f\})$$

For each state in this witness, we only show a singleton set of proof obligations, although others are possible as well (e.g., $\{\mathbf{EF}\ f, \mathbf{EF}_2\ f\}$ in state s_1). An (infinite) annotated witness to $s_1 \models \mathbf{EG\ p}$ is

$$(s_1, \{\mathbf{EG}p, p, \mathbf{EX\ EG}\ p\}) \rightarrow (s_2, \{\mathbf{EG}p, p, \mathbf{EX\ EG}\ p\}) \rightarrow s_2 \cdots$$

which moves from s_1 to s_2 and then loops infinitely on s_2. The labels are based on the identity $\mathbf{EG}\ p \Leftrightarrow p \wedge \mathbf{EX\ EG}\ p$; see Figure 2(b) for CTL identities.

An annotated witness w to $s \models \psi$ satisfies the following conditions: (1) $\pi_0 = s$, and the conjunction of the formulas in Φ_0 implies ψ: $\psi \Leftarrow \bigwedge_{\varphi_i \in \Phi_0} \varphi_i$; (2) for every state (π_i, Φ_i) in w, $\pi_i \models \varphi_i$ for each $\varphi_i \in \Phi_i$; (3) for every step $(\pi_i, \Phi_i) \rightarrow (\pi_{i+1}, \Phi_{i+1})$ in the witness, let Φ_i^t be the subset of Φ_i containing temporal operators. Then for every $\varphi_j \in \Phi_i^t$, there is $\varphi'_j \in \Phi_{i+1}$ such that $\mathbf{EX}\ \varphi'_j \Rightarrow \varphi_j$. If the witness is finite, as in the case of $\mathbf{EF}$, the proof obligation for the last step (π_k, Φ_k) does not include any temporal operators.

A sequence of annotated states is a *partial witness* if properties (1) and (2) of witnesses hold in it, and property (3) holds for every state except the last. Particularly, $(s, \{\varphi\})$ is always a partial witness for $s \models \varphi$ if this property holds in the model. Thus, using the model-checker's results cached from the computation of $[\![\varphi]\!]$, we can compute a complete witness starting from $(s, \{\varphi\})$, extending it one step at a time until either a final state is reached or a cycle can be closed. More precisely, given a partial witness with (π, Φ) as the last state, we compute the extension (π', Φ') so that (1) $\forall \varphi_i \in \Phi^t \cdot \exists \varphi'_i \in \Phi' \cdot \mathbf{EX}\ \varphi'_i \Rightarrow \varphi_i$, and (2) choose $\pi' \in Img(s) \cap \bigcap_{\varphi_i \in \Phi^t} [\![\varphi'_i]\!]$. That is, π must witness $\mathbf{EX}\varphi'_i$ for *every* temporal $\varphi_i \in \Phi$. The choice of a suitable π' is made by a *witness generation strategy* [7]. This is the (tableau-based) technique used by our model-checker XChek [6, 15], and it allows simple local specification of strategies. Clearly, other techniques are possible as well.

4.2 Strategies

Strategies are procedures for choosing which witness to show to the user, in case several are possible. A simple strategy, used by most model-checkers, is to compute the shortest possible witness (**Shortest**). For a finite witness property, such as $s \models \mathbf{EF}\varphi$, this strategy uses the identity $\mathbf{EF}\varphi \Leftrightarrow \exists i \cdot \mathbf{EF}_i\varphi$, and selects as the initial partial witness $(s, \{\mathbf{EF}_i\ \varphi\})$ by finding the least i such that $s \models \mathbf{EF}_i\ \varphi$. At each extension step, it chooses a successor for $(\pi, \{\mathbf{EF}_i\ \varphi\})$ by determining the *smallest* $j < i$ such that some $\pi' \in Img(\pi)$ satisfies $\mathbf{EF}_j\varphi$, and choosing any such π'. If $j = 0$, then $\mathbf{EF}_0\varphi$ is rewritten to the purely propositional φ, and the strategy halts successfully. This is how the witness to $s_0 \models \mathbf{EF}\ f$ in Section 4.1 was generated: $s_0 \models \mathbf{EF}_3\ f$; the least $i < 3$ than can be chosen is 2, and s_1 is the only choice. From $(s_1, \{\mathbf{EF}_2\ \varphi\})$, the least possible $i < 2$ is 1; s_2, s_3 are equally valid choices, so the strategy picks s_2 at random; finally the path is extended to s_5, which satisfies f ($\mathbf{EF}_0\ f$), and so the strategy halts.

For $\mathbf{EG}\varphi$, the strategy builds a (shortest) path until it reaches a state t which lies on a cycle: that is, there is a path from t which reaches t again, and φ holds continuously on this path. Since t satisfies $\varphi \wedge \mathbf{EX}\ \mathbf{E}[\varphi\ \mathbf{U}\ \{t\}]$, there is a finite path from t back to t, and once this has been constructed, the witness, consisting of a finite prefix followed by the cycle on t, is complete.

In this paper, we consider several strategies. We describe them informally here; for a more formal treatment, please see [7]. At each step, the set of possible successors is partitioned into *preferred* (P) and *avoided* (A); if the preferred set is nonempty, then the next state is chosen from it nondeterministically; otherwise, the next state is chosen from A. P and A can be the same throughout the construction of the witness, or can be updated. Clearly, this approach is *greedy*: decisions are made locally, and thus we may not generate the most interesting witness. Strategies with backtracking can also be defined, but their application is more expensive.

Avoid-Visited. This strategy uses the avoid set A that consists of previously visited states. The set is updated after the next state of the witness is chosen. For example, a sequence of states forming a witness to $\mathbf{EX}\ \mathbf{EF}r$ in state s_2 of the model in Figure 1(a) is s_2, s_3, or s_2, s_4 but not s_2, s_2.

Avoid-States. This strategy is similar to **Avoid-Visited**. However, it receives a set of states to avoid as a parameter, and does not update this set as the witness gets constructed.

Avoid-Conditions. This strategy is similar to **Avoid-States**, but its parameter-list consists of conditions on the next state to avoid.

Avoid-Events. This strategy receives a list L of events to avoid. Given the last state s of the partial witness, it tries to pick a successor t so that none of the events of L occur between s and t. The avoidance set stays the same throughout the witness generation process. We can further define a strategy that picks a successor that *minimizes* the number of events that fire on the transition between s and t.

Avoid-Vars. This is similar to **Avoid-Events**: given a set of variables L, the strategy extends the current partial witness by choosing the successor that does not change vari-

ables in L. We can further define a strategy that picks a successor that *minimizes* changes to variables in L.

Clearly, we can define **Prefer** counterparts of the above strategies. For example, **Prefer-Visited** extends the partial witness by preferring states which are already part of this witness.

5 Elaboration of Behavioral Models

In Section 3, we discussed specifying and validating simple scenarios. Since automatic extraction of interesting scenarios from the system is difficult (because scopes and events of interest need to be specified and because mental models are an abstraction of the behaviour of the system – they typically ignore exceptional cases), our methodology works by starting from simple scenarios that are guessed by the user, and elaborating them into more complex scenarios using *elaboration patterns*. Guessing simple scenarios is not hard – we can start just with determining that a certain event p is possible, without worrying about what caused it.

An elaboration pattern represents a typical way in which behavioral understanding moves from a set of valid and invalid scenarios – the *base scenarios* – to stronger or richer ones – the *elaborated scenarios*. This movement usually involves enriching the current vocabulary of events of interest or strengthening the relationship between the existing events. Elaboration patterns help narrow down the focus of investigation, and determine which witnesses would be most useful for elaborating the current scenario; this in turn suggests the witness-generation strategy that should be applied. Application of an elaboration pattern does *not* guarantee the existence or the utility of an elaborated scenario of the desired form.

In this section, we describe elaboration patterns which we found useful for building behavioural models. Several of these are summarized in Table 2.

Cause Weakening. Suppose we start with a validated scenario $A \rightharpoonup B$ (A causes B), whereas $A \leftharpoondown B$ (B can only happen after A) is not valid. Thus, we cannot conclude

Table 2. Elaboration patterns

Pattern	Before	Strategies	After
Cause Weakening	$A \rightharpoonup B$ $\surd$ $A \leftharpoondown B$ $\times$	**Avoid-Events**	$A \vee C \rightleftharpoons B$
Event Splitting	$A \rightharpoonup B$ $\surd$ $B = B_1 \vee B_2$ $\surd$ $A \leftharpoondown B_1$ $\surd$ $A \leftharpoondown B_2$ $\surd$ $A \rightharpoonup B_1$ $\times$ $A \rightharpoonup B_2$ $\times$	**Shortest**, **Avoid-States**	$c[A \rightharpoonup B_1]$, $c'[A \rightharpoonup B_2]$ *or* $A_1 \rightharpoonup B_1$, $A_2 \rightharpoonup B_2$, $A = A_1 \vee A_2$
Intermediate Event	$A \rightharpoonup B$ $\surd$	**Shortest**, **Avoid-Vars**	$A \rightharpoonup C \rightharpoonup B$
Intermediate Cause	$A \rightharpoonup B$ $\surd$ $A \leftharpoondown B$ $\times$		$A \leftharpoondown D$, $D \leftharpoondown B$

$A \rightleftharpoons B$. Our goal is to determine an event C such that C causes A and further $C \Rightarrow A$, so that the elaborated scenario is $A \vee C \rightleftharpoons B$. To find C, we might want to examine the causes of failure of $A \leftharpoonup B$, but the counterexample to this property is non-linear and thus may not provide the necessary understanding. Instead, we propose to examine which events other than A cause B; so the useful witnesses in this case are generated by checking $init \rightsquigarrow B$ using an **Avoid-Events**($\{A\}$) strategy. Examining these witnesses helps us guess which events need to be added to C; with each successful guess, C is increased (weakened) until we can conclude $(A \vee C) \rightleftharpoons B$.

As an example of **Cause Weakening**, consider the relationship between *landingCall*(3) and *floor*=3 in the elevator system. *landingCall*(3) $\rightharpoonup$ *floor*=3, but it is not true that *landingCall*(3) $\leftharpoonup$ *floor*=3. Using the elaboration pattern, we compute a witness to *init* $\rightsquigarrow$ *floor*=3, avoiding *landingCall*(3), which results in *init*, E_1.*liftCall*(3), E_1.*assigned*(3), E_1.*doorClosed*, E_1.*floor*=2, E_1.*floor*=3. Examining this trace allows us to identify *liftCall*(3) as another possible cause of *floor*=3. We can quickly validate that *liftCall*(3) $\rightharpoonup$ *floor*=3; and furthermore, that (*liftCall*(3) $\vee$ *landingCall*(3)) $\leftharpoonup$ *floor* $= 3$. The new event *liftCall*(3) $\vee$ *landingCall*(3) is called *call*(3), and *call*(3) $\rightleftharpoons$ *floor*=3.

Event-Splitting. Suppose we start with $A \rightleftharpoons B$, where B is a compound event $B \Leftrightarrow B_1 \vee B_2$. Thus, $A \rightharpoonup B_1 \vee B_2$ and $A \leftharpoonup B_1 \vee B_2$. We can prove $A \leftharpoonup B_1$ and $A \leftharpoonup B_2$, but neither $A \rightharpoonup B_1$ nor $A \rightharpoonup B_2$. We are interested in causes of B_1 and B_2. Potential elaborations can be some conditions under which $A \rightharpoonup B_i$, or perhaps splitting up A so that $A \Leftrightarrow A_1 \vee A_2$ and $A_1 \leftharpoonup B_1$ and $A_2 \leftharpoonup B_2$; finally, we may conclude that A leads to a non-deterministic choice between B_1 and B_2. We first examine counterexamples to $A \rightharpoonup B_i$, perhaps with the **Shortest** strategy. If this is not helpful, we suggest the following tactic: examining $A \rightsquigarrow B_1$, the existential counterpart of $A \rightharpoonup B_1$. A does not always result in B_1, but checking $A \rightsquigarrow B_1$ lets us examine cases where it does. Let V_1 be the set of states visited while generating a counterexample for $A \rightharpoonup B_1$. We can generate a witness to $A \rightsquigarrow B_1$ with the strategy **Avoid-States**(V_1); this avoids accidental similarities between paths from A to B_1 and those from A to B_2, and helps with the elaboration.

We show an application of **Event-Splitting** in the elevator system by studying the relationship between a call to floor 3 and the arrival of a given elevator to that floor. The event *floor=3* is composed of the events E_1.*floor*=3 and E_2.*floor*=3. *call*(3) $\leftharpoonup$ E_1.*floor*=3, and *call*(3) $\leftharpoonup$ E_2.*floor*=3; however, *call*(3) $\rightharpoonup$ E_1.*floor*=3 is not valid. A witness to φ=*call*(3) $\rightsquigarrow$ E_1.*floor*=3 shows a lift-call for E_2 (which is a sub-event of *call*(3)), followed by E_2.*floor*=3. We validate E_2.*liftCall*(3) $\rightharpoonup$ E_2.*floor*=3, and ask for another witness to φ, using the strategy **Avoid-Event**($\{E_2$.*liftCall*(3)$\}$). This yields the following trace: *init*, *landingCall*(3), E_2.*assignCall*(3), etc., until E_2 reaches floor 3. Thus, we observe that if both elevators are on floor 1, the assignment of calls appears to be nondeterministic.

To find a better reason, we use the **Avoid-States**(V_1) strategy, where V_1 is the set of states in the previous witness. This results in the following sequence of events: there is a call for floor 2; it is assigned to elevator 2, which moves to floor 2 to service it; there is a call for floor 3, and it is assigned to elevator 2. This allows us to make another guess:

if $E_2.\mathit{floor}$=1 and $E_1.\mathit{floor}$=2, then $\mathit{landingCall}(3)$ causes $E_1.\mathit{floor}$=3:

$$(E_1.\mathit{floor} = 1 \ \wedge \ E_2.\mathit{floor} = 2)[\mathit{landingCall}(3) \rightharpoonup E_1.\mathit{floor}{=}3]$$

Since this scenario holds, we assume that the criterion is the distance: if elevator 1 is closer to the floor called for than elevator 2, it is assigned the call. If they are equidistant, then preference is given to the elevator moving in the right direction; and otherwise the assignment is made nondeterministically by the controller. Running a sequence of witnesses using **Avoid-Visited** helps build this intuition.

Intermediate Events. Given that $A \rightharpoonup B$, is there an intermediate event C that links them, so that $A \rightharpoonup C$ and $C \rightharpoonup B$?

Intermediate Cause. This is a variant of **Cause Weakening**. Suppose we start with $A \rightharpoonup B$ valid, but $A \leftharpoondown B$ not valid. If **Cause Weakening** does not find a weaker event $A \vee C$ with $A \vee C \leftharpoondown B$, is there an intermediate event D such that D happens only because of A ($A \leftharpoondown D$), and B happens only because of D ($D \leftharpoondown B$). B can still follow A without an occurrence of D.

Variable Subset Dependence. This and the following pattern are not shown in Table 2 because they are applicable for general-purpose elaboration. The goal of **Variable Subset Dependence** is to limit the focus of the exploration. For example, we may want to study just the behaviour of the elevator E_1 by disallowing changes in variables of E_2. The pattern is to choose a subset V' of the state variables and use an **Avoid-Events**(V') strategy that attempts to avoid any changes of variables in V'.

Avoid Exception. Exceptional or error behaviour makes many systems hard to understand, but this exact understanding is usually not necessary for building mental models. For example, suppose it is possible to put elevators on service. Then most of the scenarios we attempt to validate are false: a service within the elevator would not be satisfied if the elevator is on service, scheduling of elevators to fulfill landing requests would be different, etc. This pattern allows us to exclude such behaviours from consideration. Given a failed scenario φ, we look for a condition c so that $c[\varphi]$ holds (in the elevator example, such a c is "elevator not on service"). If this fails, we can try to strengthen c by computing counterexamples to φ using **Avoid-Conditions**($\{c\}$). A similar pattern applies if the exceptional behaviour is caused by an event.

6 Evolving Models

In this section, we describe strategies for maintaining and updating mental models as the system evolves.

6.1 Formalizing Change

We start by formalizing the notion of an *evolution* of a model. We define an extension of Kripke structures which captures information about the "old" and the "new" structures; Kripke structures augmented with difference information (`diff`) are called KSDs. KSDs partition state variables into old and new and record information about

changes in transitions by associating labels to pairs of states: if there is a transition between them, it is either newly-imposed (labelled by "i") or preserved from the old structure ("p"). If there is no transition, then either the transition never existed ("n") or was deleted during the evolution ("d"). KSDs also record changes of the initial state.

Formally, a *Kripke structure with diff* (KSD) is a tuple $M = (S, s_0, s_0', R, I, I', V, V')$, where S is a set of states; s_0 and s_0' are the old and the new initial states, respectively; V and V' are sets of old and newly-added atomic propositions; $R : S \times S \to \{p, i, d, n\}$ is a labelled transition relation; $I : S \to 2^V$ and $I' : S \to 2^{V'}$ are labelling functions associating each state with the set of old and new atomic propositions, respectively, that are true in that state. In addition, for $s, x \in S$, if $I(s) = I(x)$, then s and x *used to be the same state* – they are identical but for the new variables; this is an equivalence relation, and we write $s \equiv x$. If $s \equiv x$ and $t \equiv y$, then in the old structure, transitions (s, t) and (x, y) were either both present or both absent, and thus in the new one, they are either deleted or preserved: $R(s, t) \in \{p, d\} \Leftrightarrow R(x, y) \in \{p, d\}$. The equivalence class of s under $\equiv$, $\{t \mid t \equiv s\}$, is written $\hat{s}$.

For example, we augment the model in Figure 1(a) with an additional atomic proposition q ($V' = \{q\}$). If q is true, then p does not cause r to become true. If q becomes true while r is true, r becomes false in the next state. A fragment of this model is shown in Figure 1(b). In the figure, preserved transitions are regular lines, imposed ones are extra thick, and deleted ones are dashed; those never there are not shown. The initial state of the system is now s_{13}, but $s_{13} \equiv s_0$. The transition (s_{13}, s_{14}) is considered preserved because in Figure 1(a), the transition (s_0, s_1) was present, and $s_{14} \in \hat{s}_1$, $s_{13} \in \hat{s}_0$.

Our definition of KSDs enables easy extraction of the old and the new Kripke structures. Let $M = (S, s_0, s_0', R, I, I', V, V')$ be a KSD. Then the *old Kripke structure* M_o is $(S/_{\equiv}, s_0, R_o, I, V)$, where $S/_{\equiv}$ is the set of states obtained from S via the equivalence relation $\equiv$, and $R_o(\hat{s}, \hat{t}) \Leftrightarrow \forall x \in \hat{s}, y \in \hat{t} \cdot R(x, y) \in \{p, d\}$; that is, a transition between s and t exists iff for all x, y in M whose labels agree with s and t, respectively, on old variables, the transition between x and y was either preserved or deleted. The *new Kripke structure* $M_n = (S, s_0', R_n, I \cup I', V \cup V')$ has the transition relation $R_n(s, t) \Leftrightarrow R(s, t) \in \{p, i\}$ since only the preserved and the imposed transitions are present in the new system. It is equally possible to take an old Kripke structure, and the edits (the new variables and transition changes) and compute the KSD capturing the change.

Our definition of KSDs describes the change *syntactically*. Unlike a standard simulation relation, it does not allow us to conclude anything about the *logical* relationship between the two systems; however, it does provide a way for strategies to mine the changed model for witnesses that highlight the differences induced by the change.

Note that we have not considered the *deletion* of variables. Deletion is handled by keeping the variable in V but removing dependences on this variable from the transition relation. Formally, let $M = (S, s_0, R, I, V)$ be a Kripke structure, and $x \in V$. Let s^+, s^- be states that agree on values of propositions in $V \backslash \{x\}$, but disagree on the value of x: it is true in s^+ and false in s^-. R is *independent of* x *at* s if $Img(s^+) = Img(s^-)$. R is *independent of* x if the above equality holds for all s. For example, in KSD shown in Figure 1(b), states s_{14} and s_{10} are not independent of q.

Thus, a Kripke structure can be made independent of a variable just by adding and removing transitions; this is behaviorally equivalent to removing the variable. Furthermore, removing dependence on a variable actually removes the variable from the symbolic representation of the model's transition relation.

In this paper, we do not address the problem of specifying the `diff` between the two models. However, our definition can encode many of the high-level notions of change described in the literature, e.g., the SFI feature constructs of Plath and Ryan [26].

6.2 `Diff`-Based Strategies

We define a few strategies that use change information embedded into a KSD.

Avoid-New-Variable-Events. We say that a transition (s, t) results from a *new variable event* if some proposition in V' has a different value in t as it did in s. If s is the last state of the partial witness, the preferred set consists of all successors t of s such that (s, t) does not result from new variable events. A version of this strategy that picks a transition (s, t) with the minimum *number* of new variable events can also be defined.

Avoid-New-Transitions. This strategy uses transition labels. If s is the last state of the partial witness, then t is in the preferred set if $R(s, t) = p$, and in the avoided set if $R(s, t) = i$.

Reuse-Old-Witness. The strategy is useful if the initial states of the new and the old system coincide ($s_0 \equiv s'_0$). Given a previously-generated annotated witness w for φ, with $w_i = (s_i, \Phi_i)$, this strategy prefers, at step i of generating the new witness w', states in $\hat{s}_{i+1}$.

7 Maintaining Models Under Evolution

Changes to a system can have two important effects on behavioral models: new events can be introduced and causal relationships which were established in the old system may be broken. In this section, we introduce an elaboration pattern **Exception Breaks Causation** which helps understand change and which is supported by strategies that operate on KSDs.

In the old system, $A \rightharpoonup B$ was valid. The existential counterpart $A \rightsquigarrow B$ still holds, but $A \rightharpoonup B$ no longer does. We guess that the change has introduced an exceptional condition c under which A does not lead to B, but possibly to some other event D. Our goal is to find this c,and D if it exists, so that $\neg c[A \rightharpoonup B]$ holds, and perhaps $c[A \rightharpoonup D]$ hold. Further, we may want to check whether A is necessary for D: $A \leftharpoondown D$.

Recording the difference between the two systems in a KSD allows us to use the **Avoid-New-Transitions** strategy for the counterexample to $A \rightharpoonup B$. It minimizes the dependence on the new behavior and focuses on the essential difference between the systems to help identify potential c and D. Conversely, applying **Prefer-New-Transitions** combined with **Avoid-States** allows us to compute different witnesses to $A \rightsquigarrow B$ that focus on the new behavior and yet preserve the property.

We illustrate the use of this pattern on the elevator system, which we modify by introducing a *service* feature to each elevator: once on service, it stops servicing any of

its currently-assigned landing calls and may not be assigned any other requests until it goes off service. This change breaks the scenario *landingCall*(3) $\rightharpoonup$ *floor*=3. Searching for counterexamples of this property using **Avoid-New-Transitions** yields the one where from *init*, both E_1.*service* and E_2.*service* become true, and in the next state, the landing-call for floor 3 cannot be assigned to either elevator. Further, both elevators stay on service (the last state is looping). So, we guess that the exception condition c is E_1.*service* $\wedge$ E_2.*service*.

However, c[*landingCall*(3) $\rightharpoonup$ *floor*=3] is not valid either, as the following counterexample shows: E_1 goes on service, a landing-call for floor 3 comes in, it is assigned to E_2, E_2 goes to floor 2, E_1 goes off service, and E_2 goes on service. The system may stay in this state indefinitely, without servicing the call to floor 3. Thus, we propose a weaker c, E_1.*service* $\vee$ E_2.*service*, and this guess is correct: the system behaves normally as long as neither elevator goes on service. There is no reasonable D, in this instance, with c[*landingCall*(3) $\rightharpoonup$ D]; if the elevators stay on service, then *floor*=3 may never happen, but no *positive* event which does happen instead can be identified.

To build more understanding of cases where the service feature is used but a landing-call is still being serviced, we use the **Prefer-New-Transition** strategy and examine witnesses to *call*(3) $\rightsquigarrow$ *floor*=3.

8 Tool Support

Sawblade is built on top of our symbolic model-checking tool XChek [6]. Its parts are described below.

The Vocabulary Manager keeps track of variables and events currently considered to be of interest, and hierarchical relationships between them. Elements of the vocabulary can be combined (for a more abstract event), or split up (for a more concrete one).

The Pattern Tool allows users to create behavioral models from scratch using the current vocabulary. It is similar to the corresponding part of the Bandera tool [11]; the fully-realized pattern is translated into a CTL property, which is handed to the model-checker.

The Model Manager tracks validated behavioral models and the relationships between them.

The Strategy Builder allows the user to select and customize standard strategies (such as those described in this paper). Although not currently implemented, Strategy Builder will also include a scripting language for enabling users to define their own strategies.

The Interactive Witness Generator (KEGVis) [15] uses the selected strategy to produce a witness. It can either produce it immediately, or allow manual intervention at defined breakpoints.

Sawblade can maintain Kripke structures with diff as well as ordinary Kripke structures, and construct them from a specification and an edit. This information is used whenever a change-aware strategy (such as **Avoid-New-Transitions**) is used. When a new and an old model are being examined side-by-side, all parts of the tool are aware of

it: the Vocabulary Manager marks old and newly-introduced elements, and the Model Manager indicates whether a behavioral model is validated in the old, new, or both systems. The Witness Generator distinguishes newly-introduced variables graphically when presenting witnesses, and also color-codes the types of transition used (preserved and imposed). When attempting to reuse an old witness, it can indicate the location where a removed transition made the reuse impossible.

9 Conclusions and Future Work

In this paper, we described a methodology for building compact behavioural models of existing event-driven systems. The methodology, supported by a tool Sawblade, is based on the use of model-checking for validating scenarios, and on strategy-augmented witness generation for helping elaborate these scenarios. We also described a methodology for storing information about the system evolution and using strategies that use the old and the new systems to help users understand the change in behavioural models. We illustrated our approach using an elevator controller.

In future work, we plan to augment the current capabilities of the tool by adding a scripting language, and expand the witness generator so that it can use strategies with backtracking. We are also interested in combining our methodology with query-checking [5]: once events of interest have been identified, query-checking may be effective in determining the exact relationship between them. We are also planning to provide a stronger empirical validation of our elaboration patterns.

References

1. T. Ball, A. Podelski, and S. Rajamani. "Boolean and Cartesian Abstraction for Model Checking C Programs". *STTT*, 5(1):49–58, 2003.
2. G.C. Berney and S.M. dos Santos. *Elevator Analysis, Design and Control*. IEE Control Engineering Series 2. Peter Peregrinus Ltd., 1985.
3. F. Buccafurri, T. Either, G. Gottlob, and N. Leone. "On ACTL Formulas Having Linear Counterexamples". *J. of Comp. and Sys. Sci.*, 62(3):463–515, 2001.
4. R.J.A. Buhr and R.S. Casselman. *Use Case Maps for Object-Oriented Systems*. Prentice-Hall, 1996.
5. W. Chan. "Temporal-Logic Queries". In *CAV'00*, pp. 450–463. Springer-Verlag, 2000.
6. M. Chechik, B. Devereux, and A. Gurfinkel. "χChek: A Multi-Valued Model-Checker". In *CAV'02*, 2002.
7. M. Chechik and A. Gurfinkel. "A Framework for Counterexample Generation and Exploration". in *FASE'05*, pp. 217-233, Springer-Verlag, 2005.
8. Y.-F. Chen, E.R. Gansner, and E. Koutsofios. "A C++ Data Model Supporting Reachability Analysis and Dead Code Extraction". *IEEE TSE*, 24(9), 1998.
9. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
10. E.M. Clarke, E.A. Emerson, and A.P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". *ACM TOPLAS*, 8(2):244–263, 1986.
11. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. "Bandera: Extracting Finite-state Models from Java Source Code". In *ICSE '00*, 2000.
12. M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. "Patterns in Property Specifications for Finite-state Verification". In *ICSE '99*, 1999.

13. P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Müller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. "The Software Bookshelf". *IBM Sys. J.*, 36(4), 1997.
14. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
15. A. Gurfinkel and M. Chechik. "Proof-like Counterexamples". In *TACAS'03*, pp. 160–175, 2003.
16. D. Harel and W. Damm. "LSCs: Breathing Life into Message Sequence Charts". In *FMOODS '99*, pp. 293–312, 1999.
17. C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. "Automated Consistency Checking of Requirements Specifications". *ACM TOSEM*, 5(3):231–261, 1996.
18. R. Holt. "Software Architecture as a Shared Mental Model". In *IWPC '02*, 2002.
19. G. Holzmann, D. Peled, and M.H. Redberg. "Design Tools for Requirements Engineering". *Bell Labs Tech. J.*, 2:86–95, 1997.
20. I. Jacobson, J. Rumbaugh, and G. Booch. *The Unified Software Development Process*. Addison-Wesley, 1999.
21. R. K. Keller, R. Schauer, S. Robitaille, and B. Laguë. "Pattern-Based Design Recovery with SPOOL". In *Advances in SE: Comprehension, Evaluation, and Evolution*, pp. 113–135, 2002.
22. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.
23. H.A. Müller, M.A. Orgun, S.R. Tilley, and J.S. Uhi. "A Reverse Engineering Approach to Subsystem Structure Identification". *J. of Soft. Maintenance*, 5(4):181–204, 1993.
24. G. Murphy, D. Notkin, W. Griswold, and E. Lan. "An Empirical Study of Static Call Graph Extractors". *ACM TOSEM*, 7(2), 1998.
25. G.C. Murphy, D. Notkin, and K. J. Sullivan. "Software Reflexion Models: Bridging the Gap Between Source and High-Level Models". In *FSE '95*, pp. 18–28, 1995.
26. M.C. Plath and M.D. Ryan. "SFI: A Feature Integration Tool". In *Tool Support for System Specification, Development and Verification*, Adv. in CS, pp. 201–216. 1999.
27. D. Păun and M. Chechik. "On Closure Under Stuttering". *Formal Aspects of Computing*, 14:342–368, 2003.
28. M.P. Robillard and G. C. Murphy. "Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies". In *ICSE '02*, pp. 406–416, 2002.

A Behavioral Model for Software Containers

Nigamanth Sridhar[1] and Jason O. Hallstrom[2]

[1] Electrical & Computer Engg., Cleveland State University
n.sridhar1@csuohio.edu
[2] Computer Science, Clemson University
jasonoh@cs.clemson.edu

Abstract. Software containers present an effective mechanism for decoupling cross-cutting concerns in software. System-wide concerns such as persistence, transaction management, security, fault masking, etc., are implemented as container services. While a lot of effort has been expended in developing effective container implementations, specifications for software containers are largely presented in informal natural language, which hampers predictable reasoning about the behavior of components deployed within containers. In this paper, we present a formal model for reasoning about the behavior of software containers. Our model allows developers to reason precisely about how the behaviors of software components deployed within a container are modified by the container. We further present the specifications of a few examples of container services that are found in different container implementations, and use our formal model to prove the correctness of the behavioral transformations that these services cause.

1 Introduction

A *software container* is a hosting environment for software components. It provides execution support to the components it hosts in a way that is similar to an operating system hosting processes. It also serves as a protective barrier, monitoring the interactions between hosted components and their clients, restricting the interactions to those that are deemed safe. Container-based models provide a clear separation of concerns between application logic and *enterprise services*, such as transaction management, persistence, security, etc.

Although there are several commercial container architectures, the best-recognized is the *J2EE* container provided by Sun Microsystems. The container provides a collection of enterprise services to its hosted components, selected from a predetermined set. This is not, however, the only service model to consider. Other models allow extensible service sets; some allow the service selection to change dynamically on a per-component basis [6]. We discuss some of these models in Section 2.

The services provided by a container affect *behavioral transformations* in the components it hosts. Metaphorically, the container is a type of lens: the clients' view of hosted components is altered by the services the container provides. Under this view, neither the client nor the component implementer needs to worry

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 139–154, 2006.

about implementing common enterprise services. Those services are provided by *every* component by virtue of being deployed within the container.

But there is a downside. When a component is hosted within a container, its specification no longer reflects its *total* behavior. Component clients must also consider the effects of the container services in their reasoning processes. The problem, however, is that existing container service descriptions are informal, and do not directly support formal reasoning. This is one of the key reasoning problems identified by the component-based software engineering community [5]. In the absence of more rigorous specification and reasoning techniques, the reliability of the container model may be undermined. This is the problem we address in this paper. Our contributions are:

1. A formal behavioral model of software containers, applicable across a range of commercial architectures.
2. Techniques for reasoning about the behavior of software components in the context of container-based deployment.
3. Examples of container service specifications from popular architectures, and a discussion how to use those specifications in the reasoning process.

The rest of the paper is organized as follows. In § 2, we present an overview of some existing container models. In § 3, we describe the different types of container services that can be described using our model. In § 4, we define a formal model that can be used to reason about the behavior of software containers and the services they provide. In § 5, we present some examples of services that can be specified using our model. After discussing related work in § 6, we conclude with a summary of our contributions and directions for future research in § 7.

2 Container Implementations

Software containers have been embraced as a means of modularizing cross-cutting concerns for a number of years. A container neatly encapsulates the services that cross-cut the components it hosts. Component developers need only worry about core functional concerns. In recent years, considerable effort has been invested in creating effective container implementations. We briefly examine some representative implementations.

EJB Containers. The Enterprise Java Beans (EJB) container [17] is the core of the Java 2 Enterprise Edition (J2EE) [18]. This architecture targets *enterprise class* systems. Applications are composed of EJB components that implement the business logic. The hosting EJB container provides transaction management, security services, etc. Although the component developer must abide by certain design constraints, he/she is shielded from the complexity of the underlying service implementations; the services are *container-managed*. Unfortunately, the container services are specified informally, precluding formal reasoning efforts.

Spontaneous Containers. In dynamic systems, such as those running in mobile networks, static service selection is insufficient. Popovici *et al.* [12] propose

spontaneous containers, combining container technology and dynamic aspect-oriented programming. In this model, the container is designed for application environments in which mobility and dynamic adaptation are required. A component that joins a network *discovers* available container services, and attaches itself to the appropriate services. The services are modeled as aspects that are dynamically woven into hosted components [13]. Further, the available service set can be extended dynamically. The system supports uniform service reconfiguration across hosted components without re-compilation or re-deployment. Again, formal reasoning is thwarted by a lack of formal documentation.

DRSS. Hallstrom et al. [6] describe *DRSS*, an open container architecture that allows for service variation across hosted components. The services supplied to a particular component can be applied and removed dynamically, allowing for runtime maintenance and evolution. The model is based on the notion of an *interceptor*, which processes messages flowing between component instances. Each invocation flows through an *interceptor chain*, before reaching the target component instance. Container services are implemented in the form of interceptors, making it possible for the container to inject services *between* component invocations. Since interceptor chains are allocated on a per component basis, all components need not use the same services. DRSS is also *spontaneous*, in the sense that it allows for dynamic discovery and deployment of services. Again, however, DRSS services are documented with informal descriptions alone.

As we will see, our reasoning framework applies to all these container implementations (and others), despite the variation that occurs across the models.

3 Classifying Container Services

In our formal model, we consider three classes of container services. We note that this classification is not exhaustive. For example, our model does not consider container services that ignore the specifications of hosted components. Our focus is to stay within the confines of contract-based reasoning.

1. **Monotonic addition of behavior.** This class of services are those that do not in any way modify the existing behavior of target components. They simply add functionality over and above what the components provide. Examples include *Message Logging* and *Object Visualization*.
2. **Redirection.** A service in this class remains faithful to the original specification of each hosted component C, but may redirect method invocations to an object different from the intended receiver. When a method call to a receiver object O_1 is redirected to a different object O_2, the service must meet the obligation that the new target O_2 is of the same type (or behavioral subtype) as that of O_1. Examples include *Fault Masking* and *Load Balancing*.
3. **Deferred Execution.** This class of container services may delay the delivery of a message to an object. The client may not immediately see the effect of a message sent to an object, but will see the effect at a later time. Examples include *Transaction Management* and *Batch Processing*.

4 Modeling Containers

4.1 The Need for a Formal Model

While a lot of work has gone into crafting different implementation strategies [2, 6, 7, 11, 12, 17, 18, 19], there has been relatively little effort focused on defining effective methods for reasoning formally about the behavior of components and systems that use these containers [5].

A software container acts as a mediator between a hosted component and its clients. Each message sent by a client to a hosted component is "seen" first by the container, which then sends the message to the component. Before the component receives the message, therefore, the container may have performed some actions. It may even have modified the message. Similarly, after the component has acted upon the client's message, its response to the client goes first to the container before reaching the client.

It is this "power" of the container that necessitates the need for a formal specification of its behavior. A contract that describes the behavior of a component may be useless if the component is instantiated within a container without a specification of *what* exactly the container is doing to its interactions. The specification must include details of how the specifications of hosted components are (or may be) modified by the container. Informal descriptions are not enough to achieve predictable correctness of software in the presence of containers.

4.2 The Behavioral Model

We begin our discussion of the formal model by looking at how a client may be affected by a container. When a client program makes a method invocation on a particular object, the rules of design-by-contract [10] tell us that the client must have satisfied the pre-condition of the method; and consequently, upon return from the execution of the method, the client will be able to assume that the post-condition is satisfied. This contract must not be compromised by the presence of a container. Consider a component $\mathcal{H}$ with method f_1:

$$\begin{array}{l} \textbf{Component} \quad \mathcal{H} \\ \quad \textbf{operation} \quad f_1 \\ \qquad \textbf{pre} - \textbf{condition} : \mathcal{P}_{pre} \\ \qquad \textbf{post} - \textbf{condition} : \mathcal{P}_{post} \end{array} \tag{1}$$

Further, let us suppose that this component is hosted in a container $\mathcal{C}$. When a client wants to use this component, it's view is now altered. The component that the client is using is $\mathcal{H}[\mathcal{C}] = \mathcal{C} \oplus \mathcal{H}$. We use the operator $\oplus : \mathit{Container} \times \mathit{Component} \rightarrow \mathit{Component}$ to denote the composition of a component with a software container. We use $\mathcal{H}[\mathcal{C}]$ to denote that component $\mathcal{H}$ is deployed inside container $\mathcal{C}$, and $h[\mathcal{C}]$ to denote an instance of $\mathcal{H}$ deployed in $\mathcal{C}$.

In accordance with the principles of modularity, the specification presented above must hold regardless of the context in which f_1 is invoked. The client, after the call to f_1 must be able to assume $\mathcal{P}_{post}$. Consequently, even if the component $\mathcal{H}$ is hosted in a container, this post-condition must still hold for $\mathcal{H}[\mathcal{C}]$; the

modified post-condition must be at least as strong as the original. Similarly, the client will only have to worry about satisfying the original pre-condition. If the container were to change the pre-condition, it can only be weakened.

The statement above then views a container-hosted component as a *behavioral subtype* [9] of the "*bare*" component. The container-hosted component, therefore, honors the original contract. However, it does more, and this additional behavior is *not* captured by the contract. While we can state that the new pre-condition (post-condition) is weaker (stronger) than the original, we do not have a notion of *exactly how much weaker (stronger).* Behavioral subtyping is therefore not enough for our purpose.

Model of a Container Service. Each container service $\mathcal{CS}$ contains:

- Zero or more state variables that the service may use. These variables define what the service does, and its effect on each method invocation.
- A predicate $\mathcal{M}_{pre}$ that specifies how the service modifies the pre-conditions of target methods. We call this the **pre-modifier**.
- A predicate $\mathcal{M}_{post}$ that specifies how the service modifies the post-conditions of target methods. We call this the **post-modifier**.
- The body of the service: the actions that define the functioning of the service.
- A set of additional methods that a client may invoke on a hosted component; we can view these methods as being added to the component's interface.

The minimal structure of the specification of a container service is as follows:

Definition 1. *Container Service*

Service $\mathcal{CS}$
  State : $\mathcal{S}$
  Pre_modifier : $\mathcal{M}_{pre}$
  Post_modifier : $\mathcal{M}_{post}$
  Body : $a_1; a_2, \ldots, a_n$
  Methods : $m_1, m_2, \ldots, m_k$
end Service $\mathcal{CS}$

The predicates $\mathcal{M}_{pre}$ and $\mathcal{M}_{post}$ are defined in terms of the service state variables ($\mathcal{S}$) and the *context* of each method call that the service is applied to. The context of a method call includes the name and signature of the method, the target object, and the values of all parameters to the method (if any). Upon return from a method, the context includes the current values of all the parameters, and the method's return value (if any). The methods $m_1, \ldots, m_k$ operate on the same set of variables. Encapsulation is respected; the service cannot access private fields or private methods of the target. In addition to its local state, each service has access to the following keywords:

- thismethod is a handle to the method to which the service is being applied.
- thismethod.args is the ordered sequence of parameters to thismethod. The sequence holds the input values of the parameters when referenced in

$\mathcal{M}_{pre}$, and the output values when referenced in $\mathcal{M}_{post}$. Additionally, #thismethod.args holds the input values of the parameters to thismethod in $\mathcal{M}_{post}$.
- thismethod.retval is the value returned by the method.
- target is the object that is intended as the receiver of the method call.

Each component service is *locally certifiable*. This means that we require each component service to be defined modularly and in isolation; the specification of the service is only dependent on its own local state, and is not modified by changes in other services in the container.

Model of a Service Group. Container services are aggregated into *service groups*, each of which is a sequence of container services. Components do not subscribe to a service directly; they subscribe to the service group that contains exactly the service(s) that are needed. This simplifies our reasoning model to consider only one kind of composition, without reducing expressivity.

Each service group in a container consists of the following:

- The string of container services that the service group collects. This string is referred to using the keyword services.
- The state of the service group, which is simply the fully qualified union of the states of the individual services in the service group. By fully qualified we mean that variables are of the form service-name.variable. If the same variable name is used by two services, the two variables are treated as distinct.
- A predicate $\mathcal{M}_{\mathcal{SG}\_pre}$ that specifies how the pre-conditions of target methods are modified by the service group. This predicate is the conjunction of the **pre-modifier**s of all the container services in services.
- A predicate $\mathcal{M}_{\mathcal{SG}\_post}$ that specifies how the post-conditions of target methods are modified by the service group. This predicate is the conjunction of the **post-modifier**s of all the container services in services.

Each service group is a sequence of services; the order in which the services execute is important. How then is it sufficient that the pre-modifier (post-modifier) of the service group is simply the conjunction of pre-modifiers (post-modifiers) of the individual services? Why do we not use the Hoare logic rule of sequential composition? Consider a service group $\mathcal{SG}_k$ with three services $\mathcal{CS}_1$, $\mathcal{CS}_2$, and $\mathcal{CS}_3$ with pre- and post-modifiers as follows:

$$\begin{aligned} \mathcal{CS}_1 &:: \langle \mathcal{M}_{1\_pre}, \mathcal{M}_{1\_post} \rangle \\ \mathcal{CS}_2 &:: \langle \mathcal{M}_{2\_pre}, \mathcal{M}_{2\_post} \rangle \\ \mathcal{CS}_3 &:: \langle \mathcal{M}_{3\_pre}, \mathcal{M}_{3\_post} \rangle \end{aligned} \tag{2}$$

According to the rule of sequential composition, if we had

$$\mathcal{M}_{1\_post} \Rightarrow \mathcal{M}_{2\_pre} \wedge \mathcal{M}_{2\_post} \Rightarrow \mathcal{M}_{3\_pre} \tag{3}$$

then we can say that the following about the pre- and post-modifier of $\mathcal{SG}_k$:

$$\mathcal{SG}_k :: \langle \mathcal{M}_{1\_pre}, \mathcal{M}_{3\_post} \rangle \tag{4}$$

However, since our services are all defined independently, we cannot assume such relationships as in (3) above. If $\mathcal{CS}_1$ does nothing to ensure $\mathcal{M}_{2\_pre}$ and $\mathcal{M}_{3\_pre}$, then these two predicates must be true *before* $\mathcal{CS}_1$ executes. Similarly, if $\mathcal{CS}_2$ and $\mathcal{CS}_3$ do not do anything to disrupt $\mathcal{M}_{1\_post}$, then this predicate still holds at the end of $\mathcal{CS}_3$. Hence we cannot use the rule of sequential composition to determine the pre- and post-modifier of a service group.

The formal definition of a service group is as follows. Note that the function *elements: String* $\rightarrow$ *Set* returns the elements in a string as a set.

Definition 2. *Service Group*

Service Group $\mathcal{SG}$

Modeled by: *services* : *string of* $\mathcal{CS}$

State : $\mathcal{S}_{\mathcal{SG}} = \{\forall\, cs \in elements(\mathsf{services}) : \mathcal{S}(cs)\}$

pre − modifier : $\mathcal{M}_{\mathcal{SG}\_pre} = \bigwedge_{i=1}^{|\mathsf{services}|} \mathcal{M}_{i\_pre}$

post − modifier : $\mathcal{M}_{\mathcal{SG}\_post} = \bigwedge_{i=1}^{|\mathsf{services}|} \mathcal{M}_{i\_post}$

methods : $\mathsf{methods}(\mathcal{SG}) = \{\forall\, cs \in elements(\mathsf{services}) : \mathsf{methods}(cs)\}$

When a component $\mathcal{H}$ subscribes to a service group $\mathcal{SG}$, $\mathcal{H}$ is transformed to include the state variables in $\mathcal{S}_{\mathcal{SG}}$, and all the methods in $\mathsf{methods}(\mathcal{SG})$. Moreover, the pre-condition (post-condition) of every method in $\mathcal{H}$ is transformed to include the pre-modifier (post-modifier) of $\mathcal{SG}$. For example, if the component defined in (1) subscribes to the service group in Definition 2, the component will be transformed as follows:

$$\begin{array}{l}\textbf{Component}\quad \mathcal{H}[\mathcal{SG}]\\ \quad\textbf{Additional State} : \mathcal{S}(\mathcal{SG})\\ \quad\textbf{operation}\quad f_1\\ \quad\quad\textbf{pre} - \textbf{condition} : \mathcal{P}_{pre} \wedge \mathcal{M}_{\mathcal{SG}\_pre}\\ \quad\quad\textbf{post} - \textbf{condition} : \mathcal{P}_{post} \wedge \mathcal{M}_{\mathcal{SG}\_post}\\ \quad\textbf{Additional methods:}\ \mathsf{methods}(\mathcal{SG})\end{array} \tag{5}$$

There is one more thing that we need to ensure before we can claim that this transformation is correct. Since the new pre-condition (post-condition) is the conjunction of the original pre-condition (post-condition) of the method f_1 with the pre-modifier (post-modifier) of $\mathcal{SG}$, we require the following to protect the conjunction, preventing the derivation of **false** statements:

$$\mathcal{M}_{\mathcal{SG}\_pre} \not\Rightarrow \textbf{false} \wedge \mathcal{M}_{\mathcal{SG}\_post} \not\Rightarrow \textbf{false} \tag{6}$$

We can now consider the rule required to prove that the transformation of a component by a service group is indeed correct.

Rule 1 (Behavioral Transformation)

$$\frac{\begin{array}{l}\mathcal{H} :: \langle h.f_1.\mathcal{P}_{pre}, h.f_1.\mathcal{P}_{post}\rangle \\ \{\mathcal{H}[\mathcal{SG}].f_1.\mathcal{P}_{pre}\}\ \mathcal{SG}.Body.pre\ \{\mathcal{H}.f_1.\mathcal{P}_{pre}\} \\ \{\mathcal{H}.f_1.\mathcal{P}_{post}\}\ \mathcal{SG}.Body.post\ \{\mathcal{H}[\mathcal{SG}].f_1.\mathcal{P}_{post}\}\end{array}}{\mathcal{H}[\mathcal{SG}] :: \langle h[\mathcal{SG}].f_1.\mathcal{P}_{pre}, h[\mathcal{SG}].f_1.\mathcal{P}_{post}\rangle}$$

The first antecedent requires that the specification of the method f_1 be established for the component $\mathcal{H}$ outside the container. The second antecedent requires us to show that the body of the service group when applied to the method call *on its way to* the target ($\mathcal{SG}.Body.pre$) meets the original method contract in terms of ensuring that the original pre-condition of the method f_1 holds at the end of the service execution. The third antecedent requires us to show that the body of the service group when applied to the method call *on its way back* to the client ($\mathcal{SG}.Body.post$) meets the original method contract in terms of ensuring that the original post-condition of the method f_1 holds at the end of the service execution.

Although the total number of service groups in a container is $\mid \Sigma^* \mid$ where Σ is the alphabet consisting of all the services provided by a container, some equivalences can be established among these groups. For any two service groups $\mathcal{SG}_i$ and $\mathcal{SG}_j$, if the set of services they include are the same, then the two groups are equivalent. The following is true of service groups in a container.

$$\begin{array}{l} elements(\mathcal{SG}_i.\mathsf{services}) = elements(\mathcal{SG}_j.\mathsf{services}) \quad \Rightarrow \\ \quad (\mathcal{S}(\mathcal{SG}_i) \equiv \mathcal{S}(\mathcal{SG}_j)) \wedge (\mathcal{M}_{\mathcal{SG}_i\_pre} \equiv \mathcal{M}_{\mathcal{SG}_j\_pre}) \wedge \\ \quad (\mathcal{M}_{\mathcal{SG}_i\_post} \equiv \mathcal{M}_{\mathcal{SG}_j\_post}) \wedge (\mathsf{methods}(\mathcal{SG}_i) \equiv \mathsf{methods}(\mathcal{SG}_j)) \end{array} \tag{7}$$

At any point during a method invocation, the service group is able to examine itself to determine which services have been applied so far. To do this, we use two trace variables. τ_{pre} denotes the trace of service invocations applied to a method call on its way from the caller to the callee, and τ_{post} is the trace of service invocations applied on its return from the method. Upon successful completion of a method call (when all services in $\mathcal{SG}$ have been applied, and the method has terminated), the following is true of the trace variables:

$$\begin{array}{l} \tau_{pre}(\mathcal{SG}) = \mathcal{SG}.\mathsf{services} \\ \tau_{post}(\mathcal{SG}) = reverse(\mathcal{SG}.\mathsf{services}) \end{array} \tag{8}$$

Model of a Software Container. With the pieces that we have established so far, we are now ready to describe the complete model of a software container. A software container $\mathcal{C}$ contains the following elements:

- Zero or more hosted components $\mathcal{H}_1$, ..., $\mathcal{H}_n$.
- Zero or more container services $\mathcal{CS}_1$, ..., $\mathcal{CS}_m$. These container services define the alphabet $\Sigma_{\mathcal{CS}}$ of the container $\mathcal{C}$.

– Zero or more service groups $\mathcal{SG}_1, \ldots, \mathcal{SG}_{|\Sigma^*_{\mathcal{CS}}|}$. The set of service groups is the set of all finite strings composed out of alphabet $\Sigma_{\mathcal{CS}}$.

As such, a container is modeled as a set of pairs, each mapping a hosted component to the service group that it subscribes to. $\mathbb{H}$ denotes the set of all components hosted in the container $\mathcal{C}$, and $\mathbb{SG}$ denotes the set of all service groups in the container. We model a software container, $\mathcal{C}$ as follows:

Definition 3. *Container*

$$\begin{aligned}\mathcal{C} = \{ \quad & \mathbb{H} = \{\mathcal{H}_1, ..., \mathcal{H}_n\}, \\ & \mathbb{SG} = \{\mathcal{SG}_1, \ldots, \mathcal{SG}_{|\Sigma^*_{\mathcal{CS}}|}\}, \\ & \langle \mathcal{H}_1, \mathcal{SG}_k\rangle, \langle \mathcal{H}_2, \mathcal{SG}_l\rangle, \ldots, \langle \mathcal{H}_n, \mathcal{SG}_m\rangle \quad \}\end{aligned}$$

Note: We first referred to a component $\mathcal{H}$ hosted by a container $\mathcal{C}$ as $\mathcal{H}[\mathcal{C}]$. Now, we refer to a component $\mathcal{H}$ subscribed to a service group $\mathcal{SG}$ as $\mathcal{H}[\mathcal{SG}]$. These refer to the same component — each hosted component must subscribe to exactly one service group in its hosting container.

5 Some Example Services

5.1 Message Logging

Logger (Fig. 1) is a container service that monotonically adds behavior to component methods to which the service is applied. The service does not cause any change in behavior to the original method. The service simply adds to the behavior of target methods by writing to a log the details of all calls.

The state of *Logger* consists of two strings — ILog and OLog. ILog is the log of all method invocations on their way from the caller to the target. Each element in the string is a pair consisting of a method name, and the sequence of actual parameter values passed to the named method. OLog is the log of all method invocations on their way from the target object back to the caller. OLog contains the final values of method parameters and the return value.

The pre-modifier of *Logger* adds to ILog a new pair with thismethod (method name), and the actual values of each element in thismethod.args (method arguments). @ILog here refers to the value of ILog in the state immediately preceding the start of the body of *Logger* during the target method call (*Logger.Body.pre*)[1]. The post-modifier of *Logger* creates a new pair with the name of thismethod and the sequence of return values (the actual values of elements in thismethod.args), concatenated with thismethod.retval. This new pair is added to OLog. @OLog here refers to the value of OLog in the state immediately preceding the start of *Logger* during the method's return.

Neither the pre-modifier ($\mathcal{M}^{pre}_{Logger}$) nor the post-modifier ($\mathcal{M}^{post}_{Logger}$) alter the pre- and post-conditional values of the target method's parameters. Thus, the

[1] We use the notation #x in the post-condition of a method to refer to the pre-conditional value of a variable x. The @x notation is used here to avoid confusion.

Service *Logger*
  State :
    ILog : *String of*
      ⟨ methodName : *String,*
        paramValues : *Sequence of parameter values*⟩
    OLog : *String of*
      ⟨ methodName : *String,*
        returnValues : *Sequence of return values*⟩
  pre_modifier : $\mathcal{M}^{pre}_{Logger}$
    ILog = @ILog ∗
      ⟨ thismethod.*name*,
        ⟨thismethod.args[0].*value*, . . . ,
        thismethod.args[| thismethod.args | −1].*value*⟩⟩
  post_modifier : $\mathcal{M}^{post}_{Logger}$
    OLog = @OLog ∗
      ⟨ thismethod.*name*,
        ⟨thismethod.args[0].*value*, . . . ,
        thismethod.args[| thismethod.args | −1].*value*⟩
          ∗ ⟨thismethod.retval⟩⟩
end Service *Logger*

Fig. 1. Specification of the Message Logging container service

second and third antecedents of **Rule 1** are true if the method is faithful to its behavioral specification (the first antecedent of **Rule 1**). Therefore, *Logger* causes a correct transformation.

5.2 Fault Masking

The next service we consider is one that causes the redirection of a method invocation to an object instance other than the intended receiver. In applications that provide fault tolerance, the fault masking service is very useful. The container can, in a manner that is transparent to the client, prevent method invocations from being sent to object instances that have failed. This kind of redirection only applies to components that are stateless—the method call cannot depend on the component's internal state.

The fault masking service needs the set of objects that are currently failed (R1), and for each failed object, the set of objects to which calls can be redirected (R2). There are different strategies for obtaining R1. In synchronous systems, simple timeouts can be used. In asynchronous systems, however, it is not possible to place such time bounds. We can, however, abstract away that

Service $FMask$
State :
fd : Failure detector oracle
suspects :$\{obj : obj \in \mathbb{H} : \mathsf{fd.failed}(obj)\}$
alt_objs :$Map(obj \rightarrow Set\ of\ obj)$
pre_modifier : $\mathcal{M}^{pre}_{FMask}$
target $\in$ suspects $\wedge$ alt_objs(target) $\neq \emptyset \quad \Rightarrow$
target = a_obj : a_obj $\in$ alt_objs(target) $\wedge$ a_obj $\notin$ suspects
post_modifier : $\mathcal{M}^{post}_{FMask}$ **true**
methods :
void setAlternates(a_objs: Set< $\mathfrak{T}$(target) >)
pre − condition :**true**
post − condition :
(target $\notin Keys$(alt_objs) $\Rightarrow$
alt_objs = #alt_objs $\cup$ $\{\langle$target, $a\_objs\rangle\}) \wedge$
(target $\in Keys$(alt_objs) $\Rightarrow$
alt_objs(target) = #alt_objs(target) $\cup$ $a\_objs) \wedge$
$(\forall o \in a\_objs : \mathbb{H} = \#\mathbb{H} \cup o)$
end Service $FMask$

Fig. 2. Specification of the Fault Masking container service

detail and leave it to some *failure detector* [3]. *FMask* (Fig. 2) maintains a set suspects—objects that the failure detector suspects to be failed. fd.failed(*obj*) is **true** for all suspects.

To satisfy R2, the service maintains a set of alternate objects (alt_objs) for every object *obj* in the container. This way, when the container does encounter a method call whose target is failed, it can look up an alternate object and forward the call to that object. The method setAlternates() can be invoked on a hosted object with a set of alternate objects. The argument to setAlternates() is a Set parameterized by the type of target. All objects in a_objs are added to the set $\mathbb{H}$ of container-hosted objects.

When *FMask* intercepts a method call to a suspected target, the call is directed to an object from alt_objs(target) that is still alive. If no such alternate object can be found by *FMask*, the invocation fails. On the return direction, the service does nothing, and therefore does not modify the post-condition.

Since neither $\mathcal{M}^{pre}_{FMask}$ nor $\mathcal{M}^{post}_{FMask}$ interfere with the pre- and post-condition of the target method, the second and third antecedents of **Rule 1** are true. Thus, if the original method meets its behavioral contract, *FMask* causes a correct transformation.

5.3 Transaction Management

We now come to an example of the third class of container services — deferred execution. The service we consider here is transaction management (Figure 3). Some components require that certain groups of methods be called in succession;

Service $TxnMgmt$

type Tx_Req_Types = $enum$\{Required, NotRequired\}

State :

tx_reqts :$Map(String \rightarrow$ Tx_Req_Types)

curr_txn : *String of* method

$\mu\tau$: *String of* method

pre_modifier : $\mathcal{M}^{pre}_{TxnMgmt}$

(tx_reqts(thismethod.$name$) = Required $\Rightarrow$

curr_txn = @curr_txn $*$ thismethod) $\wedge$ (target = $\perp$)

post_modifier : $\mathcal{M}^{post}_{TxnMgmt}$

(tx_reqts(thismethod.$name$) $\neq$ Required $\Rightarrow$

$\mu\tau = \#\mu\tau *$ thismethod) $\wedge$

(tx_reqts(thismethod.$name$) = Required $\Rightarrow$

expects commitTxn() $\vee$ rollbackTxn())

methods :

void setTxReqts(meth_name: String, tr: Tx_Req_Types)

pre − condition : **true**

post − condition :

(meth_name $\notin Keys(\#$tx_reqts) $\Rightarrow$

(tx_reqts = #tx_reqts $\cup$ $\{\langle$meth_name, tr$\rangle\}$)) $\wedge$

(meth_name $\in Keys(\#$tx_reqts) $\Rightarrow$

(tx_reqts(meth_name) = tr))

void commitTxn()

pre − condition : curr_txn $\neq <>$

post − condition :

curr_txn $=<>$ $\wedge$ ($\exists s$: *String of* method : $\mu\tau = s *$ curr_txn)

void rollbackTxn()

pre − condition : curr_txn $\neq <>$

post − condition :

curr_txn $=<>$ $\wedge$ ($\forall s$: *String of* method : $\mu\tau \neq s *$ curr_txn)

end Service $TxnMgmt$

Fig. 3. Specification of the Transaction Management container service

either all of these methods should succeed, or they should all fail. A partial execution may result in an inconsistent state.

A client using a component subscribed to the transaction management service must first identify the methods in the component that require transaction support. The state of *TxnMgmt* is defined in three parts:

tx_reqts: A map with the transaction requirement for each method in the component. Methods with no entry in tx_reqts do not need transaction support.

curr_txn: The string of methods that belong in the current transaction. Note that there can only be one "live" transaction at a time according to this specification; this is for simplicity of presentation due to lack of space. The specification can be extended to allow for multiple live transactions.

$\mu\tau$: The trace of all methods that target executes. This trace collects a method call when the method is actually executed, not when it is invoked. Therefore, for methods that do not need transaction support, the method is added to $\mu\tau$ upon invocation, since they are executed immediately. Methods that require transaction support are added to $\mu\tau$ when the transaction is committed.

The pre-modifier of *TxnMgmt* requires that the method being invoked be added to curr_txn if the method requires transaction support. If there is no live transaction when the target method is invoked (curr_txn $= \langle\rangle$), a new transaction is initiated. In addition, target is set to $\perp$, and control returns to the client.

The post-modifier $\mathcal{M}^{post}_{TxnMgmt}$ modifies the post-condition of the target to indicate whether it has been executed or not. The call is added to $\mu\tau$ if the method does not require transaction support. If the method does require transaction support, the method has not been executed yet; it has simply been entered into the current transaction. In this case, the post-modifier adds an **expects** clause [8] to the post-condition of the method; in the future, there has to be a call either to commitTxn() or to rollbackTxn().

In addition to the setTxReqts() method, *TxnMgmt* extends the interface of the target component with two more methods — commitTxn() and rollbackTxn(). commitTxn() can be called when there is a live transaction, and the method commits the transaction; all the methods in curr_txn are executed. This results in curr_txn showing up as a suffix of $\mu\tau$. Moreover, the post-conditional expectations of all the methods that belong in the current transaction are now met. The transaction can be rolled back using rollbackTxn(). This method simply throws away all the methods in curr_txn.

We now see how applying *TxnMgmt* to a component is a correct behavioral transformation. Assume that the correctness of the component's methods in isolation have been established (first antecedent in **Rule 1**). If a method does not require transaction support, the pre-modifier of the service does nothing to the method invocation. In case a method does require transaction support, it is not executed immediately. Instead, the *entire context* of the method is stored in curr_txn and the actual call is made when commitTxn() is called. When the method is eventually invoked, the state that the client called the method in originally is retained. In the case of a rolled-back transaction, the original target

method is never called. Thus, in all the above cases, the service does not affect the pre-condition. This proves the second antecedent in **Rule 1**.

The post-modifier of the service also does nothing in the case of methods that do not require transaction support. In the case of methods that do require transaction support, the client is required to call either commitTxn() (which will result in the post-conditions of all methods in the transaction being established) or rollbackTxn() (which will result in none of the methods being actually invoked). In all these cases, we have established the third antecedent in **Rule 1**. Thus, *TxnMgmt* causes a correct transformation.

6 Related Work

Our behavioral model of containers is related to the notion of behavioral subtyping [9]. The transformed component that the client sees as a result of the composition of the component with the container is a behavioral subtype of the original component. Our work goes beyond this, in that we are interested in specifying the additional behavior introduced by the container. Our work is loosely based upon the work on reasoning about object-oriented frameworks [14]. As in our case, Soundarajan and Fridella are interested in specifying application-specific behavior that results from specializing a framework. They use trace-based specifications to define how the template methods use hook methods to define application-specific behavior.

The work on modular aspect-oriented reasoning [4] shares similarities with our own. Clifton and Leavens show how modular reasoning techniques can be applied to understand the behavior of aspect advice. Their modifications to AspectJ in the form of *observers* and *assistants* provide a formal view of how exactly the behavior of a class is modified by the application of an aspect. The current work is derived from our previous work on modeling containers as parameterized components [16], where the container services are modeled as parameters. Our model of containers supports the dynamic addition and removal of parameters to a template. Our current approach to reasoning builds on previous work on dynamically bound parameterized components [15].

7 Conclusion

We began with the claim that the current state of container technology is an effective solution to the problem of cross-cutting concerns. Containers are used in a variety of scenarios to decouple system concerns from an application's core concerns. The software engineering community has been quick to respond to this growing demand for new and improved implementation strategies for software containers.

Although a good number of engineering issues with respect to container technology have been solved, the problem of predictable reasoning about software component behavior in the presence of software containers has not been well-studied. In this paper, we have presented a partial solution to this problem. We

have presented a formal behavioral model for software containers, along with the proof rule to show that an implementation of a service is, in fact, correct with respect to behavioral transformation.

The most important contribution of our model is the ability it affords to developers and users of software components that are deployed in a container environment to *accurately predict* the behavior resulting from composing a component with the container.

Our future plans include extending the model in ways that will allow for automated reasoning and verification of container services. We plan to build tools that will facilitate a combination of static verification and run-time monitoring to automate the reasoning process, at least partially. Our work in this direction involves extending the DRSS container architecture to include specifications. The specifications will be written in the Spec# [1] language, and will be checked at run-time to ensure that the services honor their contracts.

The work presented in this paper deals only with *behavioral* issues of container-component composition. Another direction for future research in this area involves extending our specification framework to specify *non-functional* properties of container services, such as performance, availability, etc.

References

1. M. Barnett, R. Leino, and W. Schulte. The spec# programming system: An overview. In *CASSIS 2004*, pages 49–69, 2005.
2. F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi-agent systems with a FIPA-compliant agent framework. *Software: Practice & Exp.*, (31):103–128, 2001.
3. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
4. C. Clifton and G. T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. Technical Report TR-02-04a, CS, Iowa State U., 2002.
5. I. Crnkovic, H. Schmidt, J. A. Stafford, and K. C. Wallnau. 5th ICSE cbse workshop. In *Proc. 24th ICSE*, pages 655–656, Orlando, FL, May 2002.
6. J. O. Hallstrom, W. M. Leal, and A. Arora. Scalable evolution of highly-available systems. *IEICE/IEEE Joint Special Issue on Assurance Systems and Networks*, E86-D(10):2154–2166, October 2003.
7. JBoss. The JBoss home page. http://www.jboss.org.
8. S. Kumar, B. W. Weide, P. A. Sivilotti, N. Sridhar, J. O. Hallstrom, and S. M. Pike. Encapsulating concurrency as an approach to unification. In *FSE Workshop on Specification and Verification of Component-Based Systems*, October 2004.
9. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, November 1994.
10. B. Meyer. *Design by contract*, chapter 1. Prentice Hall, 1992.
11. Object Management Group. CORBA Component Model, V3.0, 2002.
12. A. Popovici, G. Alonso, and T. Gross. Spontaneous container services. In *ECOOP 2003 – LNCS 2743*, pages 29–53. Springer, Aug 2003.
13. A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02*, pages 141–147, New York, NY, USA, 2002. ACM Press.
14. N. Soundarajan and S. Fridella. Framework-based applications: From incremental development to incremental reasoning. In *Proceedings of the 6th International Conference on Software Reuse*, pages 100–116, London, UK, 2000. Springer-Verlag.

15. N. Sridhar. *Dynamically Reconfigurable Parameterized Components.* PhD thesis, The Ohio State University, 2004.
16. N. Sridhar and J. O. Hallstrom. Generating configurable containers for component-based software. In *6th ICSE Workshop on Component-Based Soft. Eng.*, May 2003.
17. Sun Microsystems. Enterprise JavaBeans specification. http://java.sun.com/ejb/.
18. Sun Microsystems. J2EE information and specification. http://java.sun.com/j2ee/.
19. The ObjectWeb Consortium. JOnAS: Java open application server.

An Empirical Study of the Impact of Asynchronous Discussions on Remote Synchronous Requirements Meetings

Daniela Damian[1], Filippo Lanubile[2], and Teresa Mallardo[2]

[1] University of Victoria, Computer Engineering Department,
Victoria, BC, Canada
danielad@cs.uvic.ca
[2] University of Bari, Dipartimento di Informatica,
Bari, Italy
{lanubile, mallardo}@di.uniba.it

Abstract. Our research explores the combination of synchronous and asynchronous collaboration tools for global software development. In this paper we assess the impact of tool-mediated inspections to improve requirements negotiation meetings with stakeholders spread over different continents. We present the design of our investigation in an educational environment, in a course where the clients and developers in a software project were in geographically distributed locations. In particular, we studied the usefulness of asynchronous discussions in IBIS tool in enabling more effective requirements negotiations meetings. Our findings indicate that the requirements negotiations were more effective when the groups conducted asynchronous discussions prior to the synchronous negotiation meetings.

1 Introduction

Technical reviews and in particular software inspections and client reviews are considered among the most important software quality assurance techniques in software engineering. The software inspection process was first introduced by Michael Fagan at IBM [Fag76] with the main goal to find defects before testing starts and to reduce rework effort. Although its application was initially limited to code, as a complement of testing techniques, software inspections have been also applied to early life-cycle software artifacts [Lai00] because detecting defects close to their point of creation reduces rework [Boe81]. As requirements defects are the most expensive to correct if they are not detected soon, many researchers have subsequently conducted empirical studies of software inspection on requirements documents [Bas96, Bas99, Bif03, Lai02, Lan98, Por95, Sch92, The03]. Experiences from these studies indicate that inspecting requirements documents, other than producing information for correcting the document, leads to a better understanding of the real problems, increases confidence in the acquired knowledge, and improves communication among stakeholders.

The focus of our research is in this area of collaborative software development and in particular on processes that support stakeholders to collaboratively develop a shared understanding of the required software functionality. We regard requirements

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 155 – 169, 2006.

inspections and client reviews as powerful mechanisms not only for checking the requirements documentation for qualities such as completeness and correctness, but also for validating that stakeholders share same understanding of the requirements. Requirements inspections create the opportunity for identifying areas in which designers and customers need to further discuss and negotiate requirements issues.

Over the last years however, the dramatic trend towards developing software in geographically distributed settings has challenged the communication and collaboration processes in software teams [Dam03, Her03], and the development of tools and methodologies to support a combination of synchronous and asynchronous activities in distributed teams emerges as critical. In particular, it becomes important to research approaches that enable effective requirements inspections and negotiations in distributed software development, as they are activities that should support collaborative software engineering in remote teams as well as they do in traditional software teams. While research in requirements inspections and negotiations [e.g. Boe98] is being complemented by studies of inspections for validation of requirements negotiation models [Grü04, Hal03], there is little research into enabling effective negotiations that follow the identification of requirements issues during the inspections. These negotiations, as examples of requirements meetings that involve relevant project stakeholders, are traditionally difficult and expensive to coordinate, especially in geographically distributed teams.

In this paper we describe our research and early results of studying the usefulness of asynchronous discussions, as part of the requirements inspection process, to facilitate more effective synchronous requirements meetings in distributed teams. In particular, we studied the use of a web-based inspection tool, IBIS [Lan03], in support of the remote communication between clients and developers collaboratively developing a requirements specification.

IBIS supports remote teams during the inspection of requirements documents, and in particular supports teams through stages of issue Discovery, Collection, and Discrimination. During the Discovery stage, inspectors review individually the document with the help of checklists or scenarios, and records issues. In the Collection stage the inspection leader or the document's author collate recorded issues and eliminate duplicates. In the Discrimination stage the inspection team makes decisions about collated issues. The Discrimination stage is designed as a structured asynchronous discussion with two mechanisms: posting of messages for each issue under discussion and voting as to whether an issue is a true issue or not (false positives). In [Lan04], we investigated IBIS support to remote inspection teams and found that asynchronous discussions in the Discrimination stage were as effective as co-located inspection meetings at discriminating between false positives and true issues.

Our findings indicate positive impact on the effectiveness of such requirements meetings in resolving open issues when preceded by asynchronous discussions in IBIS.

The paper is structured as follows: Section 2 describes our research design, by introducing the educational environment as the context in which we conducted an empirical study of asynchronous discussions in support of synchronous requirements meetings. Section 3 then reports our early results of how IBIS was used and how we assessed the effectiveness of requirements meetings when preceded by the asynchronous meetings. We then discuss possible limitations and threats to validity and our plans for future research.

2 Research Design

To investigate the usefulness of asynchronous discussions to facilitate effective synchronous requirements negotiation meetings, we studied tool-supported remote inspections in six educational global project teams in a global software development (GSD) course. Each software project followed an iterative development process in which designers in collaboration with clients were to develop a requirements specification (RS): after a requirements elicitation stage, a requirements inspection of an early draft of RS involved the discovery as well as asynchronous discussion of requirements issues and was further followed by requirements negotiations and prototype demonstrations before the final draft of the RS was delivered. In this section we describe the research setting: the software development course, the use of IBIS and our research design that compared the effectiveness of the requirements negotiations when preceded by the asynchronous discussions in IBIS to those negotiations with no prior asynchronous discussions.

2.1 The GSD Course: Students, Groups and Remote Collaboration

The Global Software Development course was offered in a three University collaboration involving University of Victoria, Canada, University of Technology, Sydney, Australia, and University of Bari, Italy during January and May of 2005[1]. The course involved a total of 32 students. 12 of them were Master's and Doctorate students at the University of Victoria, 2 graduate and 8 undergraduate students at the University of Technology, Sydney, and 10 Master's students at the University of Bari.

As shown in **Table 1**, the Canadian students worked on software projects with the Australian and Italian groups as follows: the 12 Canadian students formed three groups of 4 (Gr1-3), the Australian students formed two groups of 5 (Gr4-5), and the Italian students formed two groups, of 3 and 7 students respectively (Gr6cl and Gr6dev). Each Canadian and Australian group was involved in two different projects, playing the role of client (C) and developer (D) respectively. Each of the two Italian groups was involved in only one project, either as a client (Gr6cl) or as a developer (Gr6dev).

Table 1. Project teams (PT) and their allocation to course projects

Country	Group	Project A (A1, A2)		Project B (B1, B2)		Project C (C1, C2)	
		PT1	PT2	PT3	PT4	PT5	PT6
Ca	Gr1	Client (C)					D
	Gr2		D	C			
	Gr3				D	C	
Au	Gr4	Developer (D)			C		
	Gr5		C			D	
It	Gr6cl						C
	Gr6dev			D			

[1] More information can be found on the course website: http://segal.cs.uvic.ca/csc576b

2.2 The Software Projects

There were three distinct projects in the course (A, B and C). Two global software project teams were allocated to each project, each with the client and developer group in two different countries (see **Table 1**). The project topics are briefly described in the following:

Project A (A1 and A2 in Table 1): Global software development system. A system to facilitate collaboration in GSD by supporting informal communication as well as document exchange in remote teams. Tasks supported by the tool included: displaying people's availability information, viewing changes between different versions of documents and authors of those changes, visualizing the evolution history of a particular document, and discovering who has been working on a particular document or section of a document.

Project B (B1 and B2 in Table 1): iMedia system. A "iMedia" software that will allow users to purchase movies online, organize their movie library, and play movies. One of the key requirements was that the interface be simple to use even for inexperienced computer users, without sacrificing key features.

Project C (C1 and C2 in Table 1): Virtual Realty system. A system that provides accurate and easy-to-find information to real estate agents and home buyers in the Victoria area. The system had to display an interactive map, where the end-user can zoom in, zoom out, pan, etc., and click on it to get the information of the property.

The projects were assigned to groups before group membership was determined. The project assignment was done so that each group worked with a different partner group for each of the two projects it was assigned (with the partner group always located in a different country), and so that the two projects it worked on were on a different topic.

2.3 An Iterative Process to Develop Requirements Specifications

Each project followed an iterative process of developing a requirements specification (RS) through collaboration between developers and clients over a period of 7 weeks. The RS development life-cycle (illustrated in **Fig. 1**) consisted of six phases of requirements discovery and validation, and through which the understanding and documentation of requirements was to be improved. Each of which stages included either client, developer or group tasks and ended with a project deliverable on which students were graded for the class. The final deliverable was the final version of the SRS, which reflected the shared understanding of the project that the clients and the developers built over the previous four phases. The project finished at the point where the developer group would start writing the code for the system called for by the project.

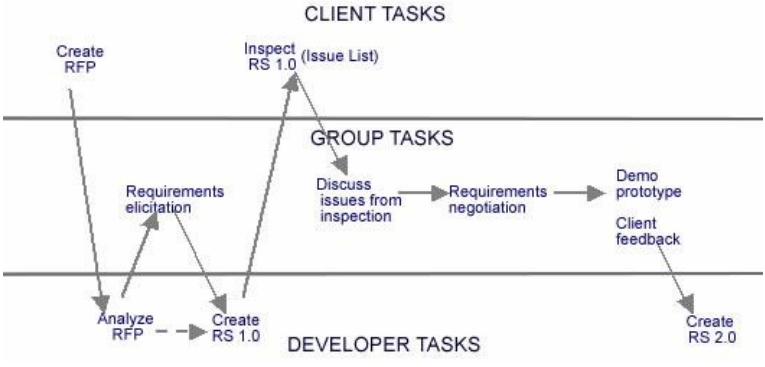


Fig. 1. An iterative process to develop the requirements specification (RS)

For each project, these six project phases consisted of:

- *Request for proposals.* Starting with the assigned project topic, the client group created a "Request for Proposals" document (RFP) which invited developers to propose their solutions to the clients' needs.
- *Requirements elicitation.* In response to the RFP, the developer group assigned to the project had three days to analyze it and come up with a list of clarifications that they needed from the clients before proceeding. The developers and the clients then held a scheduled one-hour requirements elicitation videoconference, during which the developers clarified the clients' needs and elicited more requirements. A week after the requirements elicitation meeting, the developers delivered an initial "Requirements Specification" document (RS 1.0). This document described in detail the features and scope of the project and followed the IEEE standard for requirement specification.
- *Meetingless inspection of RS 1.0 and asynchronous discussions of requirements issues in IBIS.* Upon receiving the RS 1.0 document, the clients had a week to carry on an inspection in order to identify gaps in understanding of requirements. This inspection was entirely performed online through the use of IBIS tool. With the designer team considered the authors of RS1.0, the inspection was carried out by the client team. Each member of the client team, individually, participated in the Discovery stage and read the RS 1.0 available in the system and recorded issues. The issue information contained a description of the issue found, as well as a number of issue attributes such as type severity. A course assistant collected all issues and merged duplicate issues, found by more than one client, into a unique list of collated issues. This discovery of issues was followed by a four-day asynchronous discussion. The entire project team, clients and developers, participated in this discussion using IBIS (i.e. in the Discrimination stage). The purpose of the asynchronous discussion was to come to an understanding of each issue and those issues that could be closed online (i.e. where resolution could be reached without further negotiation) or remained open issues (anything else, and which had to be further negotiated in real-time discussion). Discussants attempted to close issues by using the two mechanisms in IBIS: posting messages with respect to a certain issue, and voting as to whether it is still an open issue or is resolved and thus could be closed.

- *Requirements negotiation.* Those issues that could not be resolved during the asynchronous discussion in IBIS (i.e. open issues) were then discussed during a scheduled requirements negotiation held in a videoconference meeting between developers and clients.
- *Prototype demo.* After the requirements negotiation meetings, the developer group had one week to develop a prototype of the system to reflect the results of the negotiation. This prototype did not have to contain working code, but could consist of storyboards and paper or computer-based mockups. The purpose of the prototype was to express the developers' understanding of the project and their clients' needs, which was done through a one-hour teleconference demo. The clients could give their feedback to the developers and thus reach a consensus on the project between the two groups.
- *Create final Requirements Specification (RS 2.0).* Finally, three weeks after the prototype demos, the developers submitted a final version of the Requirements Specification (RS 2.0). This version incorporated the clients' feedback collected since the first RS draft was written, that is, through the requirements negotiation and prototype demo.

2.4 Exploring the Usefulness of Asynchronous Discussions to Facilitate Effective Synchronous Requirements Negotiations

To assess the impact of asynchronous discussions on synchronous negotiations meetings, we traced the number of open issues through the stages of each of the six projects. In particular, we studied the usefulness of asynchronous discussions prior to requirements negotiations by investigating the teams' ability to close some of the issues prior to the negotiation and focus the discussion on the issues that could not be resolved during the asynchronous discussion.

To this end, we instructed half the projects to conduct the asynchronous discussion before the negotiation, and half the projects to jump into the negotiation without asynchronous discussion. **Table 2** indicates which projects conducted the Asynchronous discussion (AD) and which did not (No AD). Then, the process variant (AD or No AD) was the main independent variable that we manipulated for experimental purposes.

When asynchronous discussions were scheduled for a project team, both clients and developers used the IBIS tool over a week, as a threaded discussion forum. The aim was to come to an understanding of each issue by exchanging messages and to an early resolution through a common agreement expressed by voting. Those open issues that could not be closed during asynchronous discussion in IBIS were then left for the synchronous negotiation meeting. For those project teams which skipped the asynchronous discussion, all collated issues were thus considered as open issues to be dealt at the negotiation.

To measure the usefulness of asynchronous discussions, we defined the following dependent variables:

- *Collated issues* = the number of open issues at the end of the inspection carried out by the client groups.

- *Closed issues during asynchronous discussion* = the number of issues for which a consensus was reached between developers and clients during the asynchronous discussion. A closed issue did not require any further discussion.
- *Open issues before sync negotiation* = the number of open issues carried over to the synchronous negotiation meeting. If there was no asynchronous discussion, open issues equate to collated issues.
- *Closed issues during sync negotiation* = the number of issues for which an agreement was reached at the videoconference requirements negotiation meeting between developers and clients.
- *Open issues after sync negotiation* = the number of issues for which an agreement has not been reached at the teleconference requirements negotiation meeting.

More specifically, to understand the impact of asynchronous discussions on the synchronous negotiations, we were interested in the variation across projects of the number of open issues resolved during the asynchronous discussions, as well as during the synchronous negotiation. To complement the quantitative data, we gathered the students' perceptions on the usefulness of the AD. In this paper we report the students' degree of agreement, based on a 4-point rating scale, to the following statements:

- "Asynchronous discrimination is useful as a preparation to the requirements negotiation meeting."
- "Reading and posting messages is effective to clear up issues."
- "Reading and posting messages is effective to develop consensus on issues."
- "Voting is effective to develop consensus on issues."

Table 2. Experimental design

project team (client/developer)	process variant
A1 (gr1/gr4)	No AD
B1 (gr2/gr6dev)	No AD
C1 (gr3/gr5)	No AD
A2 (gr5/gr2)	AD
B2 (gr4/gr3)	AD
C2 (gr6cl/gr1)	AD

3 Early Results

Here, we present the results from a preliminary analysis of the quantitative and qualitative data we collected from the IBIS database and questionnaires given to project members. We present the values on the variables we collected as well as discuss the participant's feedback with respect the usefulness of IBIS to facilitate more effective negotiations.

3.1 Effectiveness of Requirements Negotiation Meetings in Resolving Open Issues

When observed at the project level, the data sample size is too small to lend itself to statistical analysis for measuring effectiveness. Instead, we report in **Table 3** the values of the dependent variables for each of the six projects, and discuss the traces of open issues at each stage in the collaborative process as an indication of effectiveness of asynchronous discussions.

The three projects which did not conduct any asynchronous discussions (A1, B1 and C1) entered the synchronous negotiation with different numbers of open issues to be resolved: 40, 61, and 100 respectively. The number of issues that were closed during the remote meetings ranged from 26 to 47, leaving from 12 to 74 issues unresolved. At the same time, an important difference can be seen in the three projects which conducted asynchronous discussions (A2, B2 and C2); these groups entered the remote negotiation meetings with a much lower number of open issues (12, 13, 12, respectively), leading in two cases to fully resolving open issues in the meeting agenda (remained open issues 0,3 and 0 respectively). The last column in **Table 3** also shows the significant difference in the percentages of open issues after the negotiation in the projects which conducted asynchronous discussions as compared to those which did not.

Table 3. Resolution of issues from inspection to negotiation meeting

Project (cl/dev)	Collated issues	Closed issues during async discussion	Open issues before sync negotiation and percentage out of collated issues	Closed issues during sync negotiation and percentage out of collated issues	Open issues after sync negotiation	Percentage of open issues after negotiation out of open issues before negotiation
A1 (gr1/gr4)	40	No AD	40 (100.0%)	28 (70.0%)	12	30.0%
B1 (gr2/ gr6dev)	61	No AD	61 (100.0%)	47 (77.0%)	14	23.0%
C1 (gr3/gr5)	100	No AD	100 (100.0%)	26 (26.0%)	74	74.0%
A2 (gr5/gr2)	23	11	12 (52.2%)	12 (52.2%)	0	0.0%
B2 (gr4/gr3)	112	100	12 (10.7%)	9 (8.0%)	3	2.5%
C2 (gr6cl/ gr1)	23	10	13 (56.5%)	13 (56.5%)	0	0.0%

Similarly, **Fig. 2** graphically illustrates the trajectory of open issues throughout the three stages in each of the six projects, as an indication of how asynchronous discussions improved the effectiveness of remote requirements negotiations. It can be seen that all three dotted lines, representing projects with AD, finished below the three continuous lines (which correspond to projects with no AD). Particularly important is

project B2 which started with the highest number of collated issues (112) but which ends with a significantly lower number of open issues (i.e. 3), thanks to the asynchronous discussion.

To provide more insights into what actually happened during these asynchronous discussions we report in **Table 4** the intensity of message exchanging and voting, the two basic mechanisms which could be used to resolve issues before the negotiation meeting. It can be seen how participants in project B2, although with the highest number of collated issues, were nevertheless active in discussing issues (282 posted messages) and extensively exploited the voting feature provided by the tool (910 votes). We hypothesize that the intensity of the discussion made it possible to drastically reduce the number of open issues (12 left unresolved, that is 10.7%).

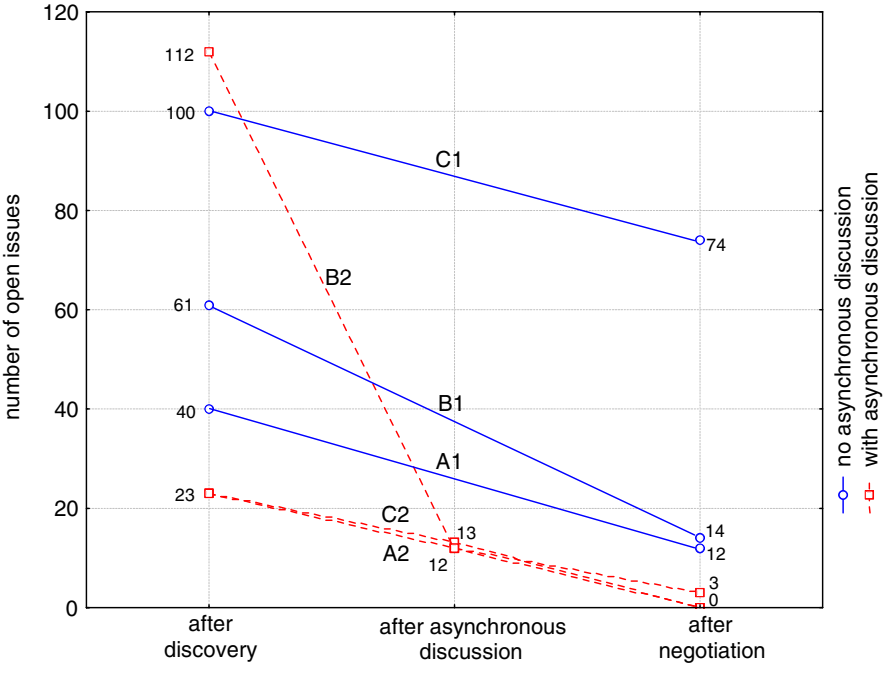


Fig. 2. Open issues at the end of three process stages

Table 4. Intensity of the asynchronous discussions

project team (cl/dev)	collated issues	participants	posted messages	messages per issue	messages per participant	votes	votes per issue	votes per participant
A2 (gr5/gr2)	23	9	131	5.7	14.6	128	5.6	14.2
B2 (gr4/gr3)	112	9	282	2.5	31.3	910	8.1	101.1
C2 (gr6cl/gr1)	23	11	72	3.1	6.5	236	10.3	21.4

3.2 Subjects' Perception

Here we present the results of the subjective evaluation of effectiveness of the asynchronous discussions. Although survey questionnaires related to the entire project experience were proposed to all students, we only considered the answers from the Canadian and Australian students because they were involved in both process variants (i.e. with and without asynchronous discussion). We analyzed answers from the Australian students (8 out of 10 responses) who experienced the asynchronous discussion as clients/reviewers and the lack of it as developers/authors. Conversely, we analyzed answers from the Canadian students (11 out of 12 responses) who experienced the asynchronous discussion as developers/authors and the lack of it as clients/reviewers.

As shown in **Fig. 3**, the great majority of students (both as clients and as developers) considered asynchronous discussion useful as a preparation to the requirements negotiation meeting. Developers, who were the authors of the requirements document under inspection, seem more enthusiastic than clients, who acted as reviewers during inspection. We believe this is due to the early feedback that developers gained as a result of the asynchronous discussion.

This is corroborated by some answers that students specified in form of further comment to the question:

"*The asynchronous discussion provided individuals the opportunity to discuss each other's issues and concerns and provide their understanding/comments on the situation in attempts for greater understanding and clarity. It helped reinforce individuals understanding of our requirements and how they work in the overall system. In this sense it helps to filter a lot of thought-to-be issues which would set the questions and agenda for the negotiation meeting*".

"*The asynchronous discussion served as an excellent platform to not only layout the issues, but also to narrow down the number of issues to be addressed in the negotiation meetings*".

Fig. 4 and **Fig. 5** show that the great majority of students appreciated to read and post messages in order to clear up issues and develop a consensus on them. Comments that provided motivation for the broad appreciation for forum-style message exchanging include:

"*Reading and posting messages during the asynchronous discussion was very effective. The question/answer style method allowed individuals to view other's interpretation of the requirements and the issues they perceive. As such this clarified much misunderstandings*".

"*By creating a written source for the asynchronous discussion, we provided the framework for the refined SRS. I appreciated the opportunity to document the issues and how they are resolved*".

However, there were also some comments highlighting limitations of asynchronous discussions:

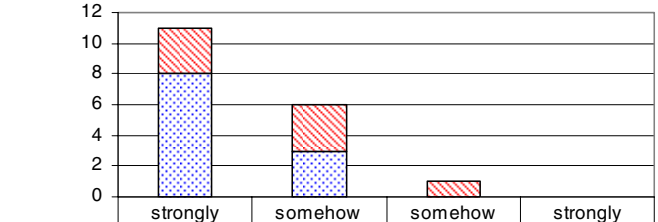
"*The messages were an effective vehicle to let both sides know where there were questions and potential disagreements. For relatively "easy" issues, it was a very effective forum. For those issue that were more involved, having many, many lengthy messages is perhaps not the best way to resolve them*".

"*This helped to understand different points of view. However, because of the time difference it took a long time to get feedback. In addition, because each item needed to be checked there was a huge amount of time spent reviewing the discussion each day*".

The other mechanism of the asynchronous discussions, voting, did not gain the same undisputed consensus as message exchanging. **Fig. 6** shows that half of the students did not acknowledge the effectiveness of voting for quickly expressing the opinion about open issues.

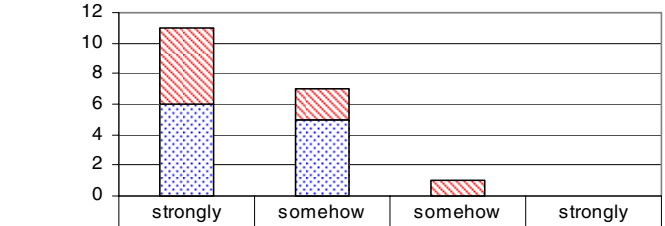
In general, students pointed out that they were asked to vote about an issue before exchanging messages. They would have preferred to vote after reading and posting messages about an issue:

"*The asynchronous discussion was useful to get everybody's opinion on the issues and to early filter out issues that all or none agreed upon. However I feel that because of time issues that we didn't get to do this stage in a properly. To have a real discussion people need to enter IBIS several times during the stage and in the setting of this project I do not know if this was done. Another problem would be that people could vote before hearing what people had to say about the issue*".



	strongly agree	somehow agree	somehow disagree	strongly disagree
client	3	3	1	0
developer	8	3	0	0

Fig. 3. "Asynchronous discussion is useful as a preparation to the requirements negotiation meeting"



	strongly agree	somehow agree	somehow disagree	strongly disagree
client	5	2	1	0
developer	6	5	0	0

Fig. 4. "Reading and posting messages is effective to clear up issues"

	strongly agree	somehow agree	somehow disagree	strongly disagree
client	1	6	1	0
developer	2	7	2	0

Fig. 5. "Reading and posting messages is effective to develop consensus on issues"

	strongly agree	somehow agree	somehow disagree	strongly disagree
client	0	4	3	1
developer	3	3	3	2

Fig. 6. "Voting is effective to develop consensus on issues"

4 Conclusions

Research advances in the area of tools and processes to support geographically distributed teams is important as developing software in global teams is becoming a fact of life nowadays. Further, empirical studies of tools and processes that support sound and proven software engineering techniques inform global teams that want to adopt practices that increase their ability to deliver high quality software. In this paper we described our research in supporting collaborative practices in geographically distributed teams, and in particular computer-mediated requirements validation techniques such as software inspections and negotiation meetings. Our strong motivation was that synchronous requirements meetings that are usually conducted during or after requirements inspections are difficult and expensive to carry out. Thus we are interested in ways in which to improve the effectiveness of such synchronous requirements meetings by allowing remote teams to discuss issues identified during the inspection in a forum of structured asynchronous discussions.

Our study has investigated the usefulness of computer-mediated, asynchronous discussions that followed requirements inspections in IBIS inspection tool, in facilitating more effective requirements negotiations that are often needed to resolve issues

from such inspections. Our findings from the observation of use of IBIS in an educational environment indicate that the teams who conducted asynchronous discussions were able to close many more issues before entering the requirements negotiation meetings. Further, the percentage of issues resolved during the synchronous meetings was much higher in those projects where the members did participate in asynchronous discussions. While these are promising research findings, we are planning a more in-depth analysis of the data we collected in this study. Directions for future research are described below, after we discuss the threats to validity in this research.

4.1 Threats to Validity

For this empirical study, we believe the following threats to internal validity that were beyond the researchers' control would make it uncertain to build cause-effect relationships between independent and dependent variables.

- *Instrumentation effects.* Instrumentation deals with the problem that differences in the results may be caused by differences in experimental material. Because in this study there were three different project topics, we cannot exclude that the topic and project complexity could have been a confounding factor.
- *Selection effects.* Results can be caused by variations in human performance. Usually, assigning subjects randomly to tasks controls this threat. In our case, selection of the participants was restricted by the practical course. For example, while Australian and Canadian students were exposed to both levels of the main independent variable, although with different roles (clients or developers), Italian students were not able to work on two projects and had the chance to choose the experimental treatment. Thus, we were not able to completely randomize the selection and participants' assignment to the different groups.

In the following we also list the most important threats to external validity, which limit the generalization of these findings to the industrial practice of distributed software development.

- *Representative subjects.* Since we involved students both as clients and as developers, they may not be representative of the population of professional stakeholders. This threat is partially mitigated by the presence of Canadian students, who were attending a specific course on global software development and then were trained on meeting protocols and negotiation techniques for requirements engineering. Some students had also previous working experience in the software business.
- *Representative artifacts.* The requirements documents inspected in this study may not be representative of industrial requirements documents. Our documents were requirements specifications for web applications while inspections are often conducted for dependable systems where quality and rework costs are perceived as critical.

4.2 Future Research

A number of important directions for furthering this research emerge as these early results indicate that the asynchronous discussions were beneficial in enabling more effective requirements negotiation meetings. To gain a more in depth understanding

of ways in which structured asynchronous discussions can support remote teams resolve open issues prior to negotiations, we are analyzing the broader context in which this causal relationship was observed. In particular, analyzing the type of issues identified during the inspections of the six teams, the complexity of those closed during the asynchronous discussion as well as negotiation meetings behavior and process will enable us to understand which factors in the computer-mediated collaborative process contributed to these results. We are conducting the analysis of the data stored in IBIS database and videotapes of the six project requirements negotiations. We hope to draw more detailed guidelines on conducting structured asynchronous discussions in support of expensive but important synchronous requirements negotiations.

Acknowledgments

We gratefully acknowledge the technical support of Luis Izquierdo and Fabio Calefato during the duration of the three-University course. Thanks also to Dr. Ban Al-Ani for her instrumental role in this collaboration and to all the students who participated to the remote projects.

References

[Bas96] V. R. Basili, S. Green , O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, and M. Zelkowitz. The empirical investigation of Perspective-Based Reading. Empirical Software Engineering: An International Journal, 1 (2): 133-164, 1996.

[Bas99] V. Basili, F. Shull, and F. Lanubile. Building Knowledge through Families of Experiments. IEEE Transactions on Software Engineering, 25(4): 456-473, July/August 1999.

[Bif03] S. Biffl, and M. Halling. Investigating the Defect Detection Effectiveness and Cost Benefit of Nominal Inspection Teams. IEEE Transactions on Software Engineering, 29 (5): 385-397, May 2003.

[Boe81] B. W. Boehm. Software Engineering Economics. Prentice Hall, Englewood Cliffs: NJ, 1981.

[Boe98] B.W. Boehm , A. Egyed, J. Kwan, D. Port, A. Shah, R. Madachy. Using the WinWin Spiral Model: A Case Study. Computer, 31(7): 33-44, July 1998.

[Dam03] D. Damian, and D. Zowghi. Requirements Engineering challenges in multi-site software development organizations. Requirements Engineering Journal, 8: 149-160, 2003.

[Fag76] M.E. Fagan, Design and code inspection to reduce errors in program development. IBM Systems Journal, 15(3), 1976, 182-211.

[Gen01] M. van Genuchten, C. van Dijk, H. Scholten, and D. Vogel, Using Group Support Systems for Software Inspections, IEEE Software, 18(3) : 60-65, 2001.

[Grü04] P. Grünbacher, M. Halling, S. Biffl, H. Kitapci, and B.W. Boehm. Integrating Collaborative Processes and Quality Assurance Techniques: Experiences from Requirements Negotiation. Journal of Management Information Systems, 20(4): 9-29, 2004.

[Hal03] M. Halling, S. Biffl, and P. Grünbacher. An economic approach for improving requirements negotiation models with inspection. Requirements Engineering Journal, 8: 236-247, 2003.

[Her03] J.D. Herbsleb, A. Mockus. An Empirical Study of Speed and Communication in Globally-Distributed Software Development. IEEE Transactions on Software Engineering, 29(3): 1-14, 2003.

[Lai00] O. Laitenberger, and J.M. DeBaud. An encompassing life cycle centric survey of software inspection. The Journal of Systems and Software, 50 (1): 5-31, January 2000.

[Lai02] O. Laitenberger, T. Beil, and T. Schwinn. An Industrial Case Study to Examine a Non-Traditional Inspection Implementation for Requirements Specifications. Empirical Software Engineering, 7(4): 345-374, December 2002.

[Lan98] F. Lanubile, F. Shull, V. Basili, "Experimenting with error abstraction in requirements documents", 5th Int. Symposium on Software Metrics (METRICS '98), pp.114-121, 1998.

[Lan03] F. Lanubile, T. Mallardo, and F. Calefato. Tool Support for Geographically Dispersed Inspection Teams. Software Process: Improvement and Practice, 8(4): 217-231, October/December 2003.

[Lan04] F. Lanubile, and T. Mallardo. A Preliminary Study On Asynchronous Discussions For Distributed Software Inspections. Proc. of the Workshop on Cooperative Support for Distributed Software Engineering Processes (CSSE 2004), September 2004.

[Per02] D.E. Perry, A.A. Porter, M.W. Wade, L.G. Votta, and J. Perpich. Reducing Inspection Interval in Large-Scale Software Development. IEEE Transactions on Software Engineering, 28(7): 695-705, 2002

[Por95] A.A. Porter, L.G. Votta, V.R. Basili. Comparing detection methods for software requirements inspections: A replicated experiment. IEEE Transactions on Software Engineering, 21(6): 563-575, 1995.

[Sch92] G. M. Schneider, J. Martin, and W. Tsai. An experimental study of fault detection in user requirements documents. ACM Transactions on Software Engineering and Methodology, 1 (2): 188-204, April 1992.

[The03] T. Thelin, P. Runeson, and C. Wohlin. An Experimental Comparison of Usage-Based and Checklist-Based Reading. IEEE Transactions on Software Engineering, 29(8): 687-704, August 2003.

Evaluation of Expected Software Quality: A Customer's Viewpoint

Krzysztof Sacha

Warsaw University of Technology, Nowowiejska 15/19,
Warszawa 00-665, Poland
k.sacha@ia.pw.edu.pl

Abstract. The paper describes an approach, which we developed to assess the expected quality of software for a huge governmental system. The assessment was done on behalf of the customer, and not of the development company, and was performed within two years along with the software development process. Our approach was based on a modification to GQM and was focused on the evaluation of the quality of methods and the deliverables of the project. The paper describes the areas of the quality evaluation, the questions that we stated and the metrics that we used. The statistical metrics defined in ISO 9126 appeared not very helpful within the context of this project.

1 Introduction

Software engineering is founded upon a set of paradigms, technologies and processes that enable the disciplined development and evolution of software systems. As with any other engineering discipline, software engineering requires methods and measurement mechanisms for evaluation of the quality of software products. Because the software development process is long and expensive, those methods and mechanisms should not only allow for the evaluation of the quality of an existing software system but also the expected quality of product under design.

Effective software quality evaluation and assurance requires models that describe what the software quality is and how can it be traced back to the development process. Two different approaches to software quality have been defined recently in a set of international ISO standards. One is focused on assuring high quality of the process by which the product is developed, while the other is aimed at a direct definition of the attributes and metrics that characterize the quality of the software product.

The requirements for a quality management system are defined in ISO 9001 [1]. All the requirements are intended for application within a software process in order to enhance the customer satisfaction, which is considered the primary measure of the software product quality. The quality management system, as defined by the standard, can be subject to a certification.

Quality characteristics of the software product are defined in ISO 9126 [2]. The characteristics are subdivide into attributes that can be measured by means of appropriate metrics. A set of metrics is defined in the accompanied technical reports [3-5]. Such definitions help in evaluating the quality of an existing software system, but gives no guidance on how to construct a high quality software product.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 170 – 183, 2006.

Unfortunately, the two standards have not been related to each other in that the quality characteristics of ISO 9126 are not referenced by ISO 9001.

A practical application of the recommendations of both of the two ISO standards needs a method for selecting the metrics and collecting data that are relevant for a particular purpose. There are a few such methods described in the literature, with the Goal Question Method approach [6-8] and the Quality Function Deployment approach [9-11] being the best known examples. All of those methods represent the viewpoint of the software development organization.

This paper describes a method, which we used to assess the expected quality of a huge software system that was developed in the years of 2003-2004 to support European Union's Common Agriculture Policy in Poland. IACS (Integrated Administration and Control System) was built in time and deployed in more than 300 regional offices. Currently, the system processes data from more than 2 000 000 farms.

The assessment was performed along with the software development process and was focused on the evaluation of the quality of methods and deliverables of the particular steps of the project. Our approach was based on a modification to Goal Question Metric approach. The modification was needed, because our assessment was done on behalf of the customer, and not of the development company, and it had no other goal in mind than just to evaluate the expected quality of the developed software. Because our customer requested full compliance with the requirements that could not be compromised, the statistical metrics defined in ISO 9126 appeared inadequate within the environment of this project.

The paper is organized as follows. Section 2 provides the reader with a short overview of the approach represented by ISO 9001, and Section 3 summarizes the approach of ISO 9126. The method that we used to evaluate the quality of the development of the IACS software is presented in Section 4, and the set of detailed questions and metrics used by the method is described in Section 5. Final remarks are gathered in Conclusions.

2 ISO 9001 Overview

ISO 9001 [1] describes the requirements for a quality management system, which is a part of the total manufacturing process. The standard is very general and applies to all types of organizations, regardless of their size and of what they do. The recommended practices can help both product and service oriented organizations and, in particular, can be used within the context of software development and manufacturing. ISO 9001 certificates are recognized and respected throughout the world.

Because of this generality it is not easy to map the recommendations of the standard into the practical activities that can be performed within a software process. Moreover, the standard is intended to be used by the manufacturers and not by the auditors that work on behalf of their customers. Therefore, it contains many recommendations that relate to resource management process, which was completely outside the scope of our evaluation. What we were expected to assess was the quality of the methods that were used by the manufacturer throughout the software development process and the quality of products of particular steps of the development: Analytical specifications, design documents, test plans and procedures, user manuals and the

resulting code. The actual implementation of code and of the testing process was also subject to our evaluation.

ISO 9001 does not define any particular model of quality. Instead, it adopts a simple approach that the quality of a product is measured by the customer satisfaction. According to this approach, no quality characteristics are defined, and the only basis for quality evaluation are the customer requirements. If those requirements are met, then the product quality can be evaluated high. The lack of a quality model makes this standard orthogonal to ISO 9126. There are no common points between the two, but also no contradiction can be found.

The top level requirement of ISO 9001 is such that a quality management system must be developed, implemented and maintained. All the processes and activities performed within the scope of this system have to be documented and recorded for the purpose of future review. A huge part of the standard relates to the processes of quality planning and management, resource management, and continuous quality monitoring, analysis and improvement. This part is not very helpful in evaluating the quality of a specific software product under design.

The part, which relates directly to the body of a software project, is a section on realization requirements. Basic requirements and recommendations that are stated therein can be summarized as follows:

1. Identify customer's product requirements, i.e. the requirements that the customer wants to meet, that are dictated by the product's use or by legal regulations.
2. Review the product requirements, maintain a record of the reviews, and control changes in the product requirements.
3. Develop the software process, clarify the responsibilities and authorities, define the inputs and outputs of particular stages.
4. Perform the necessary verification and validation activities, maintain a record of these activities, and manage design and development changes.

All of those statements are very concrete and provide valuable guidelines for auditing and evaluating the quality of a software process. Moreover, the stress that is placed on the need to meet customer requirements helps in closing the gap between the quality of the software process and the quality of software itself.

3 ISO/IEC 9126 Overview

ISO 9126 [2] is concerned primarily with the definition of a quality model, which can be used to specify the required product quality, both for software development and software evaluation. The model defines three different views of the software quality:

- Quality in use view captures the ability of a software product to help the user in achieving his or her specific goals within the specified context of use.
- External quality view captures the characteristics of a software product that can be observed when the software is executed.
- Internal quality view captures the characteristics of a software product that can be measured based on intermediate products during the software development process.

The views are related to each other in such a way that quality in use characteristics depend on the external quality characteristics, which in turn depend on the internal quality characteristics. Only the internal quality characteristics can be observed during the development process and used in order to predict the external quality and the quality in use of the final software product.

The internal and external quality models share the same set of six characteristics, which are intended to be exhaustive. All the six quality characteristics, defined in ISO 9126, are recapitulated below, along with some comments.

Functionality is defined as the ability of the software product to provide functions which meet stated or implied needs of the user. This is a very basic characteristic, which is semantically close to the property of correctness, as defined in other quality models [10]. If software does not provide the required functionality, then it may be reliable, portable etc., but no one will use it.

Efficiency is a characteristic that captures the ability of a correct software product to provide appropriate performance in relation to the amount of resources used. Efficiency can be considered an indication of how well a system works, provided that the functionality requirements are met. The reference to the amount of resources used, which appears in this definition is important, as the traditional measures of efficiency, such as the response time and throughput, are in fact system-level attributes.

Usability is a measure of the effort needed to learn and use a software product for the purpose chosen. The scope of this factor includes also the ease of assessment whether the software is suitable for a given purpose and the range of tolerance to the user errors. The features that are important within the context of usability are adequate documentation and support, and the intuitive understandability of the user interface.

Reliability is defined as the ability of software to maintain a specified level of performance within the specified usage conditions. Such a definition is significantly broader than the usual requirement to retain functionality over a period of time, and emphasizes the fact that functionality is only one of the elements of software quality that should be preserved by a reliable software product.

Maintainability describes the ease with which the software product can be analyzed, changed and tested. The capability to avoid unexpected effects from modifications to the software is also within the scope of this characteristic. All types of modifications, i.e. corrections, improvements and adaptation to changes in requirements and in environment are covered by this characteristic.

Portability is a measure of the effort that is needed to move software to another computing platform. This characteristic becomes particularly important in case of an application that is developed to run in a distributed heterogeneous environment or on a high performance computing platform, which lifespan is usually short. It is less important if the application runs in a stable environment that is not likely to be changed.

It can be noted from the above enumeration that the characteristics correspond to the product only and avoid any statement related to the development process. Each quality characteristic is very broad and therefore it is subdivided into a set of attributes. This quality model can be applied in the industry through the use of related

metrics. There are 79 internal metrics defined in [4]. The metrics are quantitative and in nearly all cases take the form of a proportion: A number of functions/items/data formats/etc. that posess certain feature, related to the the number of all functions/ items/data formats/etc. that have been defined. Despite such a mathematical definition, the evaluation of each metric is a bit subjective, because whether or not a particular function/item/data format/etc. posesses certain feature is judged by the evaluator.

4 Quality Evaluation Method

Quality evaluation methods described in the literature [6-12], represent the software development organization point of view. The values of measures collected from particular projects create a corporate memory that can help in resolving the tasks of the project planning, implementation and evaluation. Evaluation of measures can also help during the course of a project to asses the project progress and to improve those quality characteristics that are particularly important in the context of this project.

One of the most pragmatic ways to develop the set of metrics appropriate to the project is Goal Question Metric (GQM) approach described for the first time in [6] and developed since that time by NASA. GQM provides a method for transferring business goals of the development organization into a set of measurable characteristics of a software product, process or resource. The method works in a top down manner and consists of the following three steps:

1. Define business goals, typically to improve an aspect of the development.
2. For each goal define a set of questions that must be answered in order to judge whether the goal is achieved.
3. For each question define a set of metrics that provide an appropriate information for answering the question.

The steps of selecting the questions and metrics are focused on fulfilling the specific goal that defines the context for all further activities.

Quality evaluation process, which is described in this paper, was done on behalf of the customer. Planning the process we found a great difference between the quality evaluation made for and by a software manufacturer and the evaluation that was made for the customer. One difference was such that the customer had only limited access to the project data, and the quality evaluation had to be based on an evaluation of the deliverables of the software process that had been enumerated in the contract. Another difference was such that the customer had no historical data related to a set of similar projects and could not compare the actual data to the historical one. Therefore, the customer had no specific business goals, such as to improve the software process, to use less resources or to enhance the (yet unknown) software efficiency. After signing a contract it was the manufacturer who was responsible for developing the software, while the customer wanted only to be sure that everything was done right. The rationale that stood behind such a thinking was based on a hope that if things were done right, than the results would also be right. The customer was also not able (and not willing) to answer the question, which features should be evaluated. The answer we received to such a question was always the same: Check everything.

In order to fulfill the demands of our customer we had to change the top level of the GQM measurement model and replace the specific goals by a set of general subject areas that covered the development process and the set of deliverables from the process, as fully as possible. This way we decomposed the problem space into six subject areas. The first subject area referred to the development process itself, the next four areas corresponded to particular activities within the process, and the last one dealt with the software documentation. The following subject areas were defined:

1. Software process and development methods.
2. The analysis and analysis products.
3. The design and design products.
4. The implementation and the code.
5. Testing process and test documentation.
6. User manuals.

It can be noted from the above list that the areas 2 – 5 cover all the major activities (not necessarily phases) that have been identified in both: waterfall and incremental models of software development. The decomposition of the software process into the subject areas is then exhaustive with respect to the major development activities.

The evaluation of quality within a particular subject area was decomposed into an evaluation of a set of criteria, each of which defined a specific scope of judgment. Each criterion consisted of a set of closely related questions. These questions referenced object(s) within the subject area and characterized the quality issue evaluated under this criterion. The questions were answered by metrics, i.e. data that characterized the methods used to conduct the software process or the deliverables from a particular step of the software process. To avoid problems with the interpretation of data, the sets of questions were limited to the ones that could be meaningful to the customer.

A hierarchical model of the quality evaluation is shown in Figure 1.

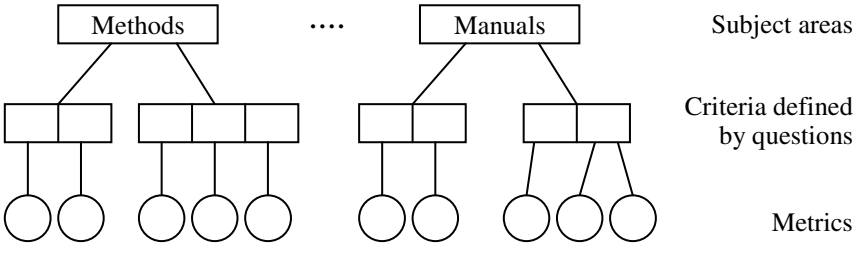


Fig. 1. Hierarchical structure of the quality evaluation model

Negative evaluation of a criterion was reported to the customer as a potential risk to the project. An advice on how to eliminate the risk was also reported to the customer as our recommendation.

5 Criteria, Questions and Metrics

In order to structure the evaluation process, we organized the set of criteria within each subject area along traceability paths: From requirements origins to the requirements specification, from the requirements specification to the design, from the design to the implementation, and from the requirements specification to the test plan. Answers to the questions within a criterion were based on objective or subjective metrics. Sample criteria, questions and metrics that were defined in particular subject areas, are discussed briefly in the next three subsections. The mechanics of the quality evaluation is described in Subsection 5.4.

5.1 Software Process and Development Methods

The main goals of work within this subject area are the identification of methods and standards that were used throughout the development process and the evaluation of how these methods were used with respect to completeness, readability and traceability of the resulting products. In order to achieve these goals we defined the following set of criteria, accompanied by the appropriate sets of questions and metrics.

Criterion M1. Methods and standards.
Questions. Which methods (standards) are used in the development process? How is the scope of these methods? Are the methods adequate in the context of this project?
Metrics. List of methods declared for the project. A mapping from the steps of the software process into the set of methods. Evaluation of the adequacy of methods.

Criterion M2. Completeness of results.
Questions. Which artifacts recommended by the methods were created? Is the set of created artifacts sufficient? Are the results documented properly?
Metrics. List of artifacts. Evaluation of the set of artifacts. A mapping from the artifacts to the volumes of documentation.

Criterion M3. Quality of the documents.
Questions. Are the created documents readable? Are the documents consistent and unambiguous? Are the documents modifiable?
Metrics. Evaluation of readability. Evaluation of consistency. Evaluation of modifiability.

Criterion M4. Traceability of the documentation.
Questions. Is the change history of the documents maintained properly? Are the subsequent documents related to the antecedent ones?
Metrics. Evaluation of the change history (versioning). Traceability graph. Evaluation of traceability between the documents.

As can be seen from the above list of criteria, the evaluation within this subject area was focused on rather formal aspects of the software process in that it did not include an in depth analysis of the contents of the documents created throughout the development process. Nearly half of the metrics were objective, which means that they depended only on the properties of objects. Other metrics, in general those that

began with the word "Evaluation", were subjective, which means that they depended not only on the properties of objects, but also on the viewpoint from which the evaluation was done.

The criteria correspond to the recommendations of ISO 9001, which state that a software process shall be developed, the outputs of particular activities shall be defined, and the results shall be recorded. The criteria have also a clear relation to the quality characteristics defined in ISO 9126, because the completeness of results together with readability, consistency, modifiability and traceability of the documentation promotes the maintainability and portability of the software product.

5.2 Analysis and Analysis Products

The main goal of work within this subject area is the evaluation of how the analysis methods were used in the project with respect to completeness, correctness and verifiability of the resulting products. There are two bunches of methods that are commonly used to perform analytical activities within the development process: Object-oriented and structured methods. Because the deliverables of both types of methods are significantly different, the quality evaluation method must be tailored to the actual analysis methodology that is used in a particular project.

The analysis performed within the IACS project was object-oriented, and relied on the use case method applied within a two-step process. In the first step, business actors and procedures were identified, and the scenarios, together with the pre- and post-conditions of these procedures were defined and documented. In the second step business procedures were refined and decomposed into sets of user functions that were to be implemented by the system. A specification of a user function included a set of alternative scenarios, a definition of exceptions and exceptional actions, and the conditions to start this particular function. The structure of data that was identified within the application domain was modeled by means of a class diagram notation.

In order to evaluate the quality of analysis and analysis products we defined the following set of criteria, accompanied by the appropriate questions and metrics.

Criterion A1. Completeness of data sources.
Questions. Which sources of information were used throughout the analysis? Was the selection of law regulations complete?
Metrics. List of sources, particularly EU and national acts, cited in the analysis documents. Evaluation of completeness.

Criterion A2. Consistency between the business model and the data sources.
Questions. Is the business model consistent with law regulations that have been identified in criterion A1?
Metrics. A mapping from the set of business procedures to the set of EU and national acts. Evaluation of consistency.

Criterion A3. Completeness of the context definition.
Questions. Is the set of input data sufficient to achieve the business goals? Is the set of output data complete with respect to business and law requirements?
Metrics. A mapping from the set of goals identified in EU and national acts to the set of data (documents) input to the business procedures. Evaluation of sufficiency. A

mapping from the set of reports defined in EU and national acts to the set of documents yielded by the business procedures.

Criterion A4. Completeness of the analysis model.
Questions. Are all of the business goals covered by the business procedures? Does the set of scenarios of each business procedure cover all types of input data?
Metrics. A mapping from the set of goals identified in EU and national acts to the set of business procedures. Evaluation of the sets of scenarios.

Criterion A5. Completeness of the functional requirements.
Questions. Are all the business procedures supported by sets of the user functions? Are all the scenarios of each business procedure covered by a set of user functions?
Metrics. A mapping from the set of business procedures to the sets of user functions. Evaluation of completeness of the coverage.

Criterion A6. Correctness of the data model.
Questions. Is the data model consistent and complete with respect to the law regulations? Does the data model comply with the best engineering practices?
Metrics. A mapping from the set of data records identified in EU and national acts to the set of classes. Evaluation of the correctness and quality of class diagrams.

Criterion A7. Usability of the user interface prototype.
Questions. Is the prototype complete? Is the prototype ergonomic?
Metrics. A mapping from the set of user functions to the sets of prototype functions. Evaluation of the ergonomics.

Criterion A8. Completeness of the non-functional requirements.
Questions. Are the non-functional requirements complete in that they define the expectations related to the security of data, performance and reliability?
Metrics. Evaluation of the non-functional requirements.

Criterion A9. Verifiability of the non-functional requirements.
Questions. Are the non-functional requirements defined in a verifiable (testable) way?
Metrics. Enumeration of these requirements that are defined in a quantitative way and those that are defined qualitatively. Evaluation of the above lists.

Criterion A10. Credibility of the verification.
Questions. Which methods of verification are used? How credible is the functional part of the test plan? How credible is the non-functional part of the test plan?
Metrics. List of verification methods. Coverage of the set of all scenarios defined within the business procedures by the test scenarios. A mapping from the set of non-functional requirements identified in criterion A7 to the set of test scenarios. Evaluation of the non-functional tests.

As can be seen from the above list of criteria, questions and metrics, the evaluation within this subject area is focused on the contents of the analytical products. The sequence of criteria moves along a path: From sources of information to business model, from business model to functions and efficiency, from functions and efficiency to verification and validation. The set of criteria is in good relation to the set of

quality characteristics defined in ISO 9126. The correspondence between the two is shown in Table 1. Portability is not included in the table, because this feature was not considered a significant premise in the environment of IACS project.

Table 1. A relation between the criteria and the quality characteristics of ISO 9126

Quality characteristic	Criteria
Functionality	A1 ... A7
Efficiency	A8, A9
Usability	A7
Reliability	A8 ... A10 (also M1)
Maintainability	A4 ... A6 (also M2 ... M4)

5.3 Testing Process and Test Documentation

The main goal of work within this subject area is to assess the credibility of the acceptance testing of the software product. Such an evaluation must cover two different aspects of the testing process:

- The quality of the test documents, i.e. test plans, test scenarios, test cases, test data and test procedures.
- The quality of the actual execution of testing with respect to the test documents and testing standards.

Acceptance testing is the process which relates the actual characteristics of the final software product to the requirements that have been stated during the requirements analysis. Therefore, the detailed structure of the test documents depends on the way in which the requirements were defined and formulated. This makes the questions and metrics that can be used for evaluation purposes also dependent on the type of the analysis methods that were used in a particular project.

The analysis of the IACS system relied on the use case method, which was applied within a two-step process of business procedures definition and user functions definition (Section 5.2). The results of the analysis were presented in the form of quite a big set of business procedures, each of which was supported by a set of user functions. The structure of the test documents reflected this structure of the analysis products.

A basic unit of testing was an application, defined as a functional module, which provided the functionality that supported a small set of closely related business procedures. The method of testing an application was defined by a test plan, which consisted of a set of test scenarios and a test procedure. A test scenario covered a single scenario of a business procedure, and consisted of a set of steps, each of which was defined by a single test case. A test case covered a scenario or a set of scenarios of a single user function, and consisted of a set of steps that corresponded to the steps of a function scenario. There was a collection of test data and a list of acceptance criteria defined for each test case.

The questions and metrics that we defined, related test scenarios and test cases to business procedures and user functions implemented by the software.

Criterion T1. Consistency of the test documents.
Questions. Is the structure of the test documentation consistent with the structure of analysis products and with the guidelines defined in the analysis documents?
Metrics. List of test documents. Evaluation of completeness of the test documents and the consistency between the test documents and the analysis products.

Criterion T2. Completeness of the functional testing.
Questions. Are all the functional requirements covered by the tests? Are all the user errors tested?
Metrics. A mapping from the set of business procedures to the set of test scenarios. A mapping from the set of user functions to the set of test cases. A mapping from the set of function scenarios to the set of test data. Coverage of business procedures by test scenarios. Coverage of user functions by test cases. Coverage of function scenarios by test data.

Criterion T3. Completeness of the non-functional testing.
Questions. Are all the non-functional requirements covered by the tests? Are the performance and stress tests included into the test plan?
Metrics. A mapping from the set of non-functional requirements (such as response times, throughput, re-start and data recovery times etc.) to the set of test scenarios. Evaluation of the test scenarios. Coverage of non-functional requirements by test data.

Criterion T4. Credibility of the acceptance criteria.
Questions. Are the acceptance criteria defined for each test data? Are the sets of acceptance criteria properly defined for each test case?
Metrics. A mapping from the set of test data to the set of acceptance criteria. A mapping from the set of test cases to the set of acceptance criteria. Evaluation of the acceptance criteria for each test case.

Criterion T5. Credibility of the testing process.
Questions. Is the test environment defined and documented properly? Is the test environment consistent and compatible with the production environment? Are the test results recorded properly? Are all the test scenarios defined in the test plan executed during the testing process?
Metrics. Evaluation of the test environment documentation. A mapping from the production environment definition (list of elements) to the test environment definition. Observation of the testing process and comparison with the test reports. A mapping from the set of test scenarios to the set of test reports. Evaluation of the test reports.

The set of evaluation criteria defined above relates to a subset of quality characteristics defined in ISO 9126. The correspondence between the two is shown in Table 2. The other three characteristics of ISO 9126 are not included in the table, because they were not verified by our team. Usability of the software was verified during the testing process directly by the testers that represented our customer. Maintainability could not be verified by means of testing. Portability was not considered a significant premise in the environment of IACS project.

Table 2. A relation between the criteria and the quality characteristics of ISO 9126

Quality characteristic	Criteria
Functionality	T1, T2, T4, T5
Efficiency	T3, T4, T5
Reliability	T3, T4, T5

The answers given by metrics to the questions stated above created the basis for a final recommendation on whether or not to accept the software product.

5.4 Evaluation Process

The software for IACS system was developed iteratively according to the guidelines of RUP – Rational Unified Process [13]. Input data to the process of quality evaluation consisted of all the documents that were created in the entire software development cycle. These included:

- Business model developed in the inception phase.
- Early analysis model of the elaboration phase.
- The set of analysis and design models created in the construction phase.
- The code and the complete set of manuals.
- Test plans and test reports (plus the observation of the testing process).

Because of the incremental nature of the development, part of the documents circulated in several versions, issued in a sequence of subsequent increments. The evaluation of the deliverables form particular increments of the process dealt mainly with new documents and other products, however, the scope of changes to the products delivered in the previous increments was also subject to investigation.

The evaluation process was decomposed into the set of subject areas listed in section 4 and was structured according to the set of criteria exemplified in sections 5.1 through 5.3. The evaluation that was done within the context of a particular criterion was guided by the set of questions and metrics.

The scope of criteria related to development methods, described in Section 5.1, was very broad, because it related to all the phases and activities of the entire development process. Therefore the evaluation of these criteria was decomposed in such a way that the questions were stated and answered separately for particular groups of artifacts. For example, the evaluation of criterion M1 (Methods and standards) was decomposed into an investigation of the software process, the analysis methods, the design methods, the implementation methods and tools, the testing methods and the documentation standards.

The answers were given to questions by metrics, only few of which were quantitative, i.e. evaluated to a numerical value. However, many metrics were formal, i.e. took the form of a mapping between the sets of artifacts or documents. Internal metrics of ISO 9126, defined as proportions, appeared useless within the environment of this project, particularly within the area of analysis. The customer required full coverage and correctness in that all the identified requirements had to be reflected in the analysis model, all the business procedures had to be specified and supported by user functions, all the data types had to be serviced, etc. 90% was not very different from

zero. An exception was the area related to testing and documentation, in which the metrics of coverage were used.

Answers to questions within a particular criterion were aggregated into a general mark within the scale of: good, satisfactory, bad, dangerous. The two lowest marks were reported to the customer as risks. The evaluation report was structured in accordance with subject areas and criteria. The results of the evaluation within a particular subject area were concluded in the form of two sections: Risks for the project and recommendations to the project. The risk section reported the bad and dangerous marks given to particular criteria within this subject area, related to the quality characteristics of ISO 9126. For example:

- The lack of functions that support certain business procedures creates the risk that the required functionality of software will not be met.
- The lack of readable design models creates the risk that the maintainability of software will be unacceptable low.

The recommendation section advised on what to do in order to avoid the risks identified in the evaluation process and described in the previous section. Recommendations related to the risks listed above could read, e.g.:

- Define the functionality that is missing.
- Create the models that are missing and improve readability of those that exist.

6 Conclusions

This paper describes a practical method that can be used to evaluate the expected quality of software under design. The evaluation process does not refer directly to the existing standards, however, it is consistent with the definitions of the quality models of both ISO 9001 and ISO 9126. The mechanics of the evaluation is based on a set of criteria that are decided by stating questions and finding answers to those questions. The collection of criteria is structured into a set of subject areas that cover the set of activities or phases that exist in the most popular software processes.

The method was used successfully in evaluating the expected quality of software developed for a huge governmental system. The evaluation was performed on behalf of the customer and not the manufacturer of the system. The criteria, questions and metrics that helped in answering the questions allowed for a systematic, in depth analysis of the deliverables of the particular development activities. As result, several risks that could have a negative impact on the quality of the resulting software were revealed and identified. The recommendations helped the customer in avoiding these risks. IACS system was build and certified for use within the deadline.

The advantages of the method can be summarized as follows:

- The method can be tailored to any particular software process or method that can be used in the development of software.
- The application of the method leads to such results, i.e. to the evaluation of criteria, that are readable and meaningful to the customer.

- Negative evaluation of particular criteria can easily be translated into risk warnings and recommendations on what to improve in order to enhance the expected quality of the final product.

The method is simple in use, does not relay on any historical data, and need not be supported by a computerized tool.

References

1. ISO 9001: Quality management systems – Requirements. ISO (2001)
2. ISO/IEC 9126-1: Software engineering – Product quality – Part 1: Quality model. ISO/IEC (2001)
3. ISO/IEC TR 9126-2: Part 2: External metrics. ISO/IEC (2001)
4. ISO/IEC TR 9126-3: Part 3: Internal metrics. ISO/IEC (2001)
5. ISO/IEC TR 9126-4: Part 4: Quality in use metrics. ISO/IEC (2001)
6. Basili, V.R., Weiss, D.M.: A Methodology for Collecting Valid Software Engineering Data, IEEE Trans. Software Eng., 6 (1984) 728-738
7. Basili, V.R., Caldiera, G., Rombach, H.D.: The Goal Question Metric Approach. In: Encyclopedia of Software Engineering, Wiley-Interscience, New York (1994)
8. Solingen, R., Berghout, E.: The Goal/Question/Metric Method, McGraw-Hill (1999)
9. Erikkson, I., McFadden, F.: Quality Function Deployment: A Tool to Improve Software Quality. In: Information & Software Technology, 9 (1993) 491-498
10. Fenton, N: Software Metrics: A Rigorous Approach, Chapman and Hall (1993)
11. Haag, S., Raja, M.K., Schkade, L.L.: Quality Function Deployment Usage in Software Development. In: Communications of the ACM, 1 (1996) 41-49
12. Lethbridge, T.C., Sim, S.E., Singer, J.: Studying Software Engineers: Data Collection Techniques for Software Field Studies, Empirical Software Engineering, 10 (2005) 311-341
13. Kruchten, P.: Rational Unified Process: An Introduction, Addison Wesley (2003)

Using Design Metrics for Predicting System Flexibility*

Robby, Scott A. DeLoach, and Valeriy A. Kolesnikov

Department of Computing and Information Sciences, Kansas State University,
234 Nichols Hall, Manhattan, Kansas, USA
{robby, sdeloach, valkov}@ksu.edu

Abstract. While multiagent systems have been extolled as dynamically configurable and capable of emergent behavior, these qualities can be a drawback. When the system changes so that it no longer achieves its goals, emergent behavior is undesirable. Giving agents the autonomy to adapt and then expecting them to adapt only in acceptable ways requires rigorous design analyses. In this paper, we propose metrics for determining system flexibility at design time. Our approach is based on organization-based multiagent systems, which allows multiagent systems to adapt within a preset structure. We tailored the Bogor model checker to efficiently analyze the adaptive behaviors of these systems and to determine their properties such as fault-tolerance and cost-efficiency. We develop state-space coverage metrics to allow designers to make informed trade-offs at design-time between computational cost and system flexibility.

1 Introduction

Distributed systems that can adapt to dynamically changing environments are becoming prevalent. The advent of the Internet and wireless communications has allowed users to expect the ability to integrate their local applications with data and computational capabilities from any location, at any time. Applications for distributed, adaptive systems include information systems, communication systems, sensor networks, and cooperative robotic teams. The prevailing approach to building these distributed, adaptive systems is that of multiagent systems in which locally autonomous agents coordinate with each other to provide access to distributed information and services. The power in the multiagent approach is that, because of autonomy, the agents can adapt to their environment and thus satisfy their assigned goals.

While multiagent systems have been widely touted as dynamically configurable and capable of emergent behavior, this has also been noted as a significant drawback. Most designers/users are not comfortable with the idea of pure emergent behavior where agents learn or discover and continually modify their behavior. As long as the behavior being learned or discovered is consistent with system goals, emergent behavior is not a problem. However, when the system functionality changes to where it no longer accomplishes its stated goals, emergent behavior becomes undesirable.

* This material is based upon work supported by the National Science Foundation under Grant No. 0347545 and by the Air Force Office of Scientific Research.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 184 – 198, 2006.

A key problem faced by the Agent-Oriented Software Engineering (AOSE) community is ensuring that multiagent systems will actually perform as desired without undesirable emergent behavior, which results from individual agent autonomy. Giving agents the autonomy to adapt and then expecting them to adapt only in acceptable ways requires rigorous analyses when designing and building these systems. In this paper, we propose some new design metrics and investigate one in depth for determining multiagent system flexibility at the design level. Our approach is based on previous work on organization-based multiagent systems [7] and model checking [6]. We describe how a software model checking framework such as Bogor [15] can be customized to efficiently analyze emergent behaviors of multiagent systems.

The novelties and the main contributions of our work are: (1) efficient state-space exploration of multiagent system behaviors at the design level, (2) mining the constructed state-spaces to determine their desirable/undesirable properties such as fault-tolerance and cost-efficiency, (3) proposing several useful design metrics based on state-space coverage measures to capture these properties, and (4) validating the predictions from the proposed metrics by using simulation methods. By using the proposed metrics, we believe system designers are better equipped to make informed trade-off between cost and effectiveness of multiagent systems, as well as preventing ineffective system designs.

The paper is organized as follows. Section 2 presents a motivating example used to illustrate our approach. Section 3 presents the multiagent organization design metamodel that we consider. Section 4 presents an efficient state-space exploration technique implemented using the Bogor framework. Section 5 presents some of our proposed metrics that we validate in Section 6 using simulation methods. Section 7 presents some related work. Finally, Section 8 concludes and presents some future work.

2 Motivating Example

Throughout this paper we use an example from cooperative robotics to demonstrate our model of organization-based multiagent systems and the application of our design metrics. A simplified cooperative robotics example is used (due to space constraint), however it is still interesting enough to illustrate the application of the organization metamodel and the effect of the loss of hardware capabilities to the system.

The example we use is the Cooperative Robotic Floor Cleaning Company (CRFCC). Essentially, we are designing a team of robots whose goal is to clean the floors of a building. At initialization, the team is given a map of the building including the type of flooring of each area. The floors may be tiled or carpeted and may be littered with large debris as well as small dirt particles that must be cleaned. Therefore, the CRFCC must be able to pick up any large objects and then vacuum or mop the floors, based on their type. The team should be able to clean the floors of the building even when faced with failures of individual robots or specific capabilities on those robots. This implies that the team must be able to (1) *assign* floor areas based on individual team member's capabilities (i.e. to mop, vacuum, sweep, etc.), (2) *recognize* when a robot is incapable of carrying out its responsibilities, and (3) *reorganize* the team to allow the team to achieve its goal in spite of individual failures.

3 Organizational Metamodel

To allow teams of agents (or robots) to adapt to their environment by determining their own organization at runtime, we developed a metamodel that describes the knowledge required to define and reason about an organization [7, 14]. Given this knowledge, we have shown that multiagent teams are able to organize (and reorganize) themselves in an attempt to adapt to dynamic environments.

Organizations are typically defined as a set of agents who play roles within a structure that defines the relationships between those roles [3]. In our organization metamodel shown in Fig. 1 (simplified due to space constraint; we refer the readers to [7, 14] for a more complete description), we include these basic concepts of goals (G), roles (R), and agents (A), plus agent capabilities (C) and a set of assignments (Φ).

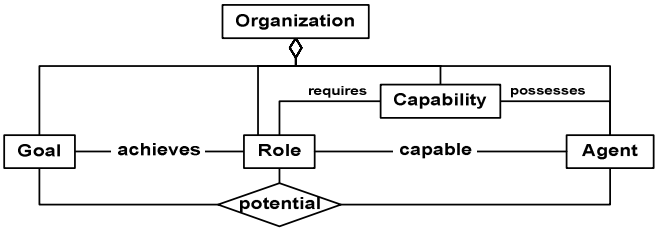


Fig. 1. Organization Metamodel (simplified)

3.1 Goals

Every organization is designed with a specific purpose or goal. In our metamodel, each organization has a set of goals, G, that it seeks to achieve in support of a top-level goal g_o, which we define as a desired end state. G is derived by decomposing g_o into a tree of sub-goals that describe how g_o can be achieved. Following the KAOS goal based requirements modeling approach [18], we allow goals to be decomposed into a set of non-cyclic sub-goals using either AND-refinement or OR-refinement. Eventually, g_o is refined into a set of leaf nodes, denoted by G_L, that are actually achieved by agents in order to achieve g_o. The active goal set, G_A (where $G_A \subseteq G_L$), is the set of goals that an organization is trying to achieve at the current time.

In order to provide an ordering for goal achievement, we define a precedence relation between goals. We say that goal g_1 *precedes* goal g_2 if g_1 must be achieved before g_2 can be achieved, which allows the team to work on a subset of the leaf goals, thus reducing the size of G_A. The initial active goal set, G_{A0}, consists of all leaf goals without predecessor goals. However, G_A changes as goals are achieved; achieved goals are removed from the active goal set and new goals are inserted. We denote a sequence of active goal sets G_A' as $G_A' = [G_{A1}, G_{A2}, \ldots, G_{An}]$.

The goal model for the CRFCC is shown in Fig. 2. Goals are denoted as specialized class components using the <<Goal>> notation. Conjunctive sub-goals are connected to their parents by a diamond shaped connector ($\Diamond$) while disjunctive sub-goals are connected to their parent by a triangle shaped connector (Δ). Goals can have

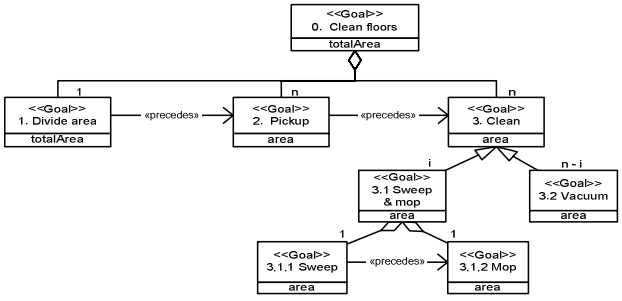


Fig. 2. Goal Model (simplified)

parameters. The *totalArea* parameter refers to the entire area to be cleaned. Since total area may include tile and carpeted areas, the team divides it into sub-areas (denoted by the *area* parameters) to be tackled independently. However, to ensure the entire task is completed as efficiently as possible, the team must consider the capabilities of its team members when partitioning the areas and assigning areas to robots. The multiplicity *n* represents the total number of sub-areas while *i* refers to the number of tiled areas. The <<precedes>> notation indicates precedence relation between goals.

The goal model consists of five leaf goals: Divide Area, Pickup, Sweep, Mop, and Vacuum. The precedence relations provide the natural ordering that is required to clean the floors. The *n* sub-areas must be created before work may begin; this results in *n* Pickup goal instances being created as well as *i* Sweep and Mop goals and *n-i* Vacuum goals. Due to the precedence relation, the individual areas must be picked up and any large debris removed before the areas can be swept, mopped, or vacuumed. Finally, depending on what type of flooring is present, the areas are either (1) swept and then mopped, or (2) vacuumed.

3.2 Capabilities

Capabilities are the key to determining exactly which agents can be assigned to what roles in the organization. Currently, we view a *capability* as an atomic entity used to define the abilities of agents. Capabilities can capture *soft* abilities such as the ability to access resources, communicate, migrate, or computational algorithms. They also capture *hard* capabilities such as those of hardware agents such as robots, which include sensors and effectors. In the CRFCC example, the robots must have specific capabilities to carry out the cleaning operation. Thus, we assume the capabilities shown in Table 1 are available for designing CRFCC robots.

The *org* capability is a reasoning ability that allows a robot to divide the current search area up into *n* areas based on the type of flooring (as well as other possible factors such as size, wall placement, etc.). The *search* capability allows robots to move about an area and identify items that need to be picked up before cleaning can begin. This capability is actually a combination of low-level capabilities such as movement and sensing as well as reasoning abilities to identify target items based on shape, size, color, etc. The *move* capability refers to the ability of a robot to pickup an item and to move it out of the way for cleaning. This capability could be representa-

Table 1. CRFCC Capabilities

Name	Description
org	Ability to logically divide the area between team members
search	Ability to search an area for large debris
move	Ability to move large debris
sweep	Ability to sweep a tiled area
mop	Ability to mop a tiled area
Vacuum	Ability to vacuum a carpeted area

Table 2. RM0 Roles

Name	Required Capabilities	Leaf Goals Achieved
Organizer	org	1. Divide Area
Pickuper	search, move	2. Pickup
TileCleaner	sweep, mop	3.1.1. Sweep & 3.1.2. Mop
Vacuummer	vacuum	3.2. Vacuum

tive of robotic arms or gripper devices. The last three capabilities, *sweep*, *mop* and *vacuum* are straightforward capabilities that also require integration of low-level capabilities. These capabilities provide the ability to clean tile and carpeted floors.

3.3 Roles

Each organization has a set of roles R that it can use to achieve its goals. A role defines the capabilities required for an entity to achieve a goal (or set of goals) in the organization. The *achieves* function ($R \times G_L \rightarrow [0,1]$) tells how good a role is for realizing a specific goal (0 = no ability to achieve the goal, 1 = excellent ability to achieve the goal); if agent A is *better* at attaining goal G than agent B, we would expect that *achieves(A,G)* > *achieves (B,G)*. However, to be assigned to play a role, agents must have a sufficient set of capabilities to play that role. Thus, agents *possess* capabilities while roles *require* a certain set of capabilities. The set of capabilities required by a role is captured using the *requires* relation ($R \times C$).

For the CRFCC example, we developed two sets of roles, or *role models*, that the individual robots can play in order to accomplish the overall CRFCC goal. In the first role model (RM0), we attempted to combine basic capabilities to carry out specific goals. For RM0 we came up with four roles as shown in Table 2. In this case, we would need a robot with the org capability to be assigned to the Organizer role in order to achieve the initial goal, Divide Area. Once the area was divided into sub-areas, the robots with the search and move capabilities would be assigned to play the Pickuper role to achieve all the Pickup goals generated for each sub-area. Once this goal was achieved, robots with sweep and mop capabilities would be assigned to the *TileCleaner* role to achieve goals Sweep and Mop for each tiled sub-area while robots with the vacuum capability would be assigned to play the *Vacuummer* role to achieve the Vacuum goal for each carpeted area.

In a second version of the role model, Role Model 1 (RM1) as shown in Table 3, we took a slightly different approach to defining the roles for the CRFCC. Instead of defining roles to carry out basic functions in the application, we defined a role for each leaf goal. Essentially, we divided the *TileCleaner* role into *Sweeper* and *Mopper*.

Table 3. RM1 Roles

Name	Required Capabilities	Leaf Goals Achieved
Organizer	org	1. Divide Area
Pickuper	search, move	2. Pickup
Sweeper	sweep	3.1.1. Sweep
Mopper	mop	3.1.2. Mop
Vacuummer	vacuum	3.2. Vacuum

3.4 Agents

The organization metamodel also includes a set of heterogeneous agents, *A*. For our purposes, *agents* are computational system instances that inhabit a complex dynamic environment, sense, and act autonomously in this environment, and by doing so realize a set of goals. Agents are assigned specific roles in order to achieve organizational goals. The current set of possible assignments of agents to a role is captured by the *potential* function ($G_L \times R \times A \rightarrow$ [0,1]). The range of the *potential* function indicates how well an agent can play a role and how well that role can achieve the goal, based on the *achieves* and the *capable* scores.

However, the *potential* function does not indicate the actual assignment of agent *a* to role *r* to achieve goal *g*, it simply defines possible assignments. To capture the actual assignments, we define an *assignment set* Φ, which consists of goal-role-agent tuples, $<g,r,a>$. If $<g,r,a> \in \Phi$, then agent *a* has been assigned by the organization to play role *r* in order to achieve goal *g*. As discussed above, however, only agents with the right set of capabilities may be assigned to a role. To capture a given agent's capabilities, we define a *possesses* function ($A \times C \rightarrow$ [0,1]), whose dynamic value ranges from no (0) capability to an excellent (1) capability. Using a role's required capabilities and the capabilities possessed by an agent, we compute the ability of an agent to play a given role, which we capture in the *capable* function (A x $R \rightarrow$ [0,1]).

4 Using Bogor to Explore Behaviors of Multiagent Organization

Bogor [4, 15] is a model checking framework designed for extensibility to enable more effective incorporation of domain knowledge into verification models and model checking algorithms. In contrast to most existing model checkers, Bogor's modeling language (BIR) provides constructs commonly found in modern programming languages including dynamic object and thread creation, garbage collection, virtual method calls and exception handling. This rich modeling language has enabled us to model check relatively large concurrent Java programs. In addition, BIR can be extended with new primitive types, expressions, and commands associated with a particular domain (e.g., multi-agent systems, avionics, security protocols, etc.) and a particular level of abstraction (e.g., design metamodels, design models, source code, byte code, etc.) to enable efficient modeling and state-space representation. Furthermore, Bogor's well-organized module facility allows new algorithms (e.g., for state-space exploration, state storage, etc) and new optimizations (e.g., heuristic search strategies, domain-specific scheduling, etc.) to be easily swapped in to replace Bogor's default model checking algorithms. To support effective BIR software model

```
system OrganizationMetamodel {
  extension Set for SetModule {
    typedef type<'a>;
    expdef Set.type<'a> create<'a>('a ...);
    actiondef add<'a>(Set.type<'a>, 'a);
  } …
extension AOM for AOMModule {
    typedef Agent; typedef Goal; typedef Role;
    expdef boolean isTopGoalAchieved(Set.type<Goal> goals);
    expdef Goal chooseGoal(Set.type<Goal>);
    expdef Role chooseRole(Goal goal);
    expdef Agent chooseAgent(Role role);
  }
  active thread Search() {
    Goal g; Role r; Agent a;
    Set.type<Goal> achievedGoals;
    Set.type<Triple.type<Goal, Role, Agent>> assignments;
    achievedGoals := Set.create<Goal>();
    assignments := Set.create< Triple.type<Goal, Role, Agent>>();
    while (!AOM.isTopGoalAchieved(achievedGoals)) do
      g := AOM.chooseGoal(achievedGoals);
      r := AOM.chooseRole(g);
      a := AOM.chooseAgent(r);
      Set.add(achievedGoals, g);
      Set.add(assignments, Triple.create(g, r, a));
    end
  }
}
```

Fig. 3. Organization Metamodel and Search Algorithm in BIR (excerpts)

checking, we have extended well-known optimization/reduction strategies [8, 16] such as collapse compression [11], data [12] and thread [5] symmetry, partial-order reduction [6] strategies that leverage static/dynamic escape and locking analyses.

We leverage BIR's extensibility to represent the organization metamodel presented in the previous section, as shown in Fig. 3. Each entity in the metamodel (e.g., agents) is modeled as a (native) first-class type in BIR (e.g., Agent). Similarly, we define auxiliary structures such as tuple and set and their corresponding abstract operations to enable more concise model. Moreover, by modeling organization entities and data structures as first-class type in BIR, we can instruct Bogor to use customized state representations better suited to the analysis' level of abstraction. For example, we leverage symmetric property of set to efficiently store set instances in the state-space representation (e.g., {Agent1, Agent2} = {Agent2, Agent1}). Accordingly, first-class abstract operations are implemented as an extension of the model checker instead of being a part of the model itself, thus, they are interpreted in the model checker's space instead of the model's space. This is analogous to adding new native types and instructions in a processor. That is, we can use the new types and instructions to better represent and more efficiently execute programs instead of representing them using a limited set of types and instructions. ([15] describes how to implement Bogor extensions.) The extension module AOM requires an organization instance as a Bogor configuration that contains information such as the goal structure, functions, and relations described in the previous section for that particular instance. Given the configuration, Bogor exhaustively explores the state-space of the BIR model in Fig. 3 for the

specified organization instance. That is, the BIR model is reusable for any model instance of the organization metamodel specified in Fig. 1.

We now describe the extensions used in Fig. 3: (1) `isTopGoalAchieved`: given a set of achieved goals, the extension determines whether the goal set can satisfy the requirement of achieving the top goal of an organization by looking at its goal structure, (2) `chooseGoal`: given a set of achieved goals, the extension non-deterministically chooses the next goal to be achieved. The extension leverages the *precedes* relation such that it does not choose goals whose preceding goals are not in the achieved goal set, which reduces the number of paths during the state-space explorations. In other words, `chooseGoal` non-deterministically chooses a goal from the active goal set G_A. The user can also specify to optimize paths on disjunctive goals as an option, i.e., by preventing to choose a goal whose disjunctive sibling goals are already achieved, (3) `chooseRole`: given a goal, the extension non-deterministically chooses the role that can achieve the goal based on the organization *achieves* function (i.e., when *achieves* gives a non-zero value), and (4) `chooseAgent`: given a role, the extension non-deterministically chooses the agent that can assume that role based on the organization *capable* function.

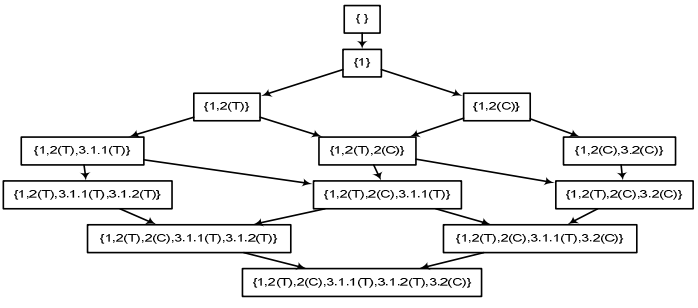


Fig. 4. Goal Achievement State-space (**G**) for CRFCC Example in Fig. 2

The `Search` thread explores all possible assignment sets that satisfy an organization's top goal. For optimization, we only store states in the beginning of each iteration of the `Search`'s loop. Fig. 4 presents the model's goal achievement state-space of the organization in Fig. 2 (without disjunctive goal optimization). The graph is generated based only on the goal structure (without considering roles and agents); Bogor can generate several state-spaces, for example, on goal ($\mathcal{G}$), goal-role ($\mathcal{GR}$), and goal-role-agent ($\mathcal{GRA}$). Each node in the figure represents a set of goals that has been achieved, and each edge represents an achievement of a goal. Note that each node in $\mathcal{G}$ implicitly represents the active goal set G_A, i.e., the set of goal achievements represented by the outgoing edges; thus, each path captures the sequence of active goal set G_A'. For goals that may be achieved at the same time, we follow the usual concurrency interleaving model that represents two transitions t_1 and t_2 that are executed at the same time as two paths $t_1 \rightarrow t_2$ and $t_2 \rightarrow t_1$. In the case where an organization cannot achieve the top goal, Bogor can give an empty (or a partial) state-space.

The CRFCC organization's goal diagram in Fig. 2 has a parameter n, which is the number of area that the agents have to clean up. Based on several experiments that we

have on varying *n*, we concluded that for the kinds of analyses that we perform, we do not actually need to have the actual concrete numbers of area; it is enough to focus on the characteristic of each area, i.e., whether it is tiled or carpeted. The reason is that we cannot actually distinguish between the areas at the design level; thus, for example, one carpeted area is the same as another carpeted area, i.e., one goal achievement sequence of one carpeted area would be similar to the other carpeted area's. Therefore, we only divide the area into two logical categories: carpeted (C) or tiled (T). This approach is akin to symmetry reduction [12] techniques usually used in model checking (e.g., the symmetric set mentioned previously), i.e., using one representative to reason about a set of entities that share the same properties. Section 8 describes extrapolation methods to recover the actual achievement sequences.

The key property of our analysis is that the state-space represents all possible ways to achieve the top goal even in the presence of agent failures/retries and malfunctions/recoveries. That is, an agent may retry several times before actually achieving a goal, or an agent malfunctions completely at some point in time, hence, the rest of the goals must be achieved by some other agents. In the end, if we take an actual system trace that achieves the organization top goal (with failures/retries and malfunctions), and if we project a sequence of the actual goal achievements for that trace, that sequence is in the state-space constructed by our analysis. For instance, let us consider the edge $\{\}\rightarrow\{1\}$. This edge actually represents any system trace prefix that eventually achieves 1. For example, an agent A can be assigned to achieve 1 and then it somehow malfunctions without completing it, the system then reorganizes and assigns a different agent B to the goal. After several attempts, that B finally achieves 1. In a goal-agent state-space, this trace is represented by a path with prefix $\{\}\rightarrow\{<1, B>\}$ (and without A contributing to goal achievements in the path's suffix).

5 Design Metrics

Based on the analysis results presented in the previous sections, we have developed a set of metrics that can be used at design time to measure system performance. Specifically, in this paper we focus on a set of metrics based on path coverage in an attempt to measure the flexibility of the system. We define *system flexibility* as the ability of the system to reorganize to overcome individual agent failures. Ideally, such a metric would be unambiguous, simple to compute, and produce a small set of values that allows the designer to directly compare a set of possible system designs.

There are several pieces of coverage information that can be mined from the different state-spaces generated by Bogor. To measure system flexibility, we compare the state spaces of $\mathcal{G}$ and $\mathcal{GRA}$ for particular organization designs. Based on this approach, we have proposed the following metrics:

- **Covering Percentage:** For each path in **G**, we determine whether there exists a path in the **GRA**. For covering, we compute the percentage of paths in **G** that are covered in **GRA** (higher is better).
- **Coarse Redundancy:** For each path in **G**, we determine the number of paths in **GRA** (or **GR**) that cover it and give a coarse redundancy rate (paths in **GRA** divided by paths in **G**; higher is better).

There are other statistics that can be mined from the state-spaces (Section 7 describes more metrics). For example, given two GRAs (or GRs) of two different organization instances with the same goal diagram (hence, the same $\mathcal{G}$), we are able to compare coverage differences of the two with respect to $\mathcal{G}$. These coverage metrics allow designers to explore different role models and agent models for a given goal structure.

6 Metric Validation

We created four predefined robot teams to validate the metric set proposed in the previous section; we designed four different robot teams implementing RM0 and RM1 as defined in Section 3 for the goal model of Fig. 2. The robot teams were designed to provide a wide range of capabilities while keeping the same number of robots on each team at five. Each agent was given the capabilities to carry out exactly one of the application's leaf goals. The specific capabilities given to each robot are shown in Table 4. We want to predict and to compare the flexibility of each system.

Table 4. Agent System Designs

Name	AS0	AS1	AS2	AS3
A1	org, search, move, vacuum, sweep, mop	org, search, move, vacuum	org, search, move	org
A2	org, search, move, vacuum, sweep, mop	search, move, vacuum, sweep	search, move, vacuum	search, move
A3	org, search, move, vacuum, sweep, mop	vacuum, sweep, mop	vacuum, sweep	vacuum
A4	org, search, move, vacuum, sweep, mop	org, sweep, mop	sweep, mop	sweep
A5	org, search, move, vacuum, sweep, mop	org, search, move, mop	org, mop	mop

Table 5. Bogor Coverage Results

Organization (# paths in **G** = 10)	Coarse Redundancy (G-GRA) Rate	
	RM0	RM1
AS0	15625	15625
AS1	324	729
AS2	16	64
AS3	.3	1

We applied our analysis to the agent system designs; Table 5 presents Bogor analysis results for AS0-3 with RM0-1. For the experiments, we used an Opteron 248 workstation, Linux OS, and Java 5.0 (64-bit) with maximum heap of 256 MB; all the state-space analyses for $\mathcal{G}$ and $\mathcal{GRA}$ finished under 15 seconds (combined). All systems achieve 100% covering of $\mathcal{G}$ as there are agents that can achieve each goals (if a goal model has disjunctive sub-goals, it is possible to create organizations that can achieve the overall goal without agents that can achieve all disjunctive sub-goals). Based on the numbers, Bogor predicts that RM1 is more flexible than RM0, AS0 is the most flexible system, AS3 is the least flexible, and AS1 is more flexible than AS2.

To empirically evaluate the flexibility of designs AS0 – AS3 on the role models RM0 and RM1, we developed a simulation that stepped through the CRFCC applica-

tion. To measure the flexibility, we simulated capability failure. At each step in the simulation, a randomly selected assigned goal was achieved. Then, one robots capability was randomly selected and then tested to see whether or not it had failed. Based on a predefined *capability failure rate* (0 – 100%), we determined whether or not the selected capability had failed. For simplicity of presentation we used a single failure rate; however, the model could easily be extended to handle different failure rates. In addition, in contrast to the coarse redundancy metric that takes into account the possibility of agents to recover from a failure, we assumed once failed, a capability remained failed for the life of the system. Then, reorganization was performed to assign available robots to available goals and to de-assign robots if their capability had failed, and they were no longer able to play their assigned role.

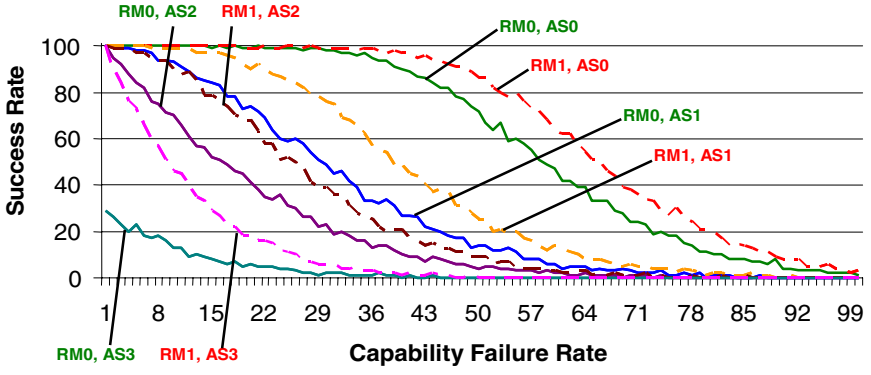


Fig. 5. Comparison of Role Model 0 vs. Role Model 1 for Agent Sets AS0 - AS3

Using a floor with 10 separate areas, we simulated each system (AS0 – AS3) on each role model (RM0 and RM1). For each role model, system combination was simulated for failure rates ranging from 0 to 100% for 1000 system executions. To compare the effectiveness of the role models using the four agent system designs, we looked at the results using each of the agent systems above. The results in Fig. 5 show that Role Model 1 provides more flexibility than Role Model 0. Furthermore, the simulation results confirm that AS0 is the most flexible while AS3 is the least one, and AS1 is more flexible than AS2. Note that the curve for (RM0, AS3) does not start at 100% since AS3 does not have an agent capable of playing the TileCleaner role.

The Bogor predictions and the simulation results make sense because: (1) in contrast to RM1, not all agents can assume the *TileCleaner* role in RM0, (e.g., A4 and A5 in AS3), (2) AS0 is the most flexible because each agent in AS0 can achieve any goal, (3) AS3 is the least flexible because each of its agents can assume at most one role, and (4) AS1 is more flexible than AS2 because AS1 agents have more capabilities.

6.1 Tradeoff Analysis

To demonstrate the usefulness of our metric in making design decisions, consider the following situation. Assume we have already developed a system based on RM1 and AS2, but now want to upgrade our system with a fixed budget. Our engineers deter-

mined that we could either (1) buy a single additional robot with three capabilities, or (2) buy five additional capabilities and integrate them onto our current robots. Essentially, option 2 equates to upgrading from AS2 to AS1 while option 1 would produce, for example, AS5 (option 1a) or AS6 (option 1b) as shown in Table 6.

Bogor's analysis results indicate that option 2 is better with a coarse redundancy rate of 729. The coarse redundancy rates for both option 1a and 1b are 216 while the original system (AS2) had a coarse redundancy rate of 64. Thus, using the coarse redundancy metric, we would choose option 2.

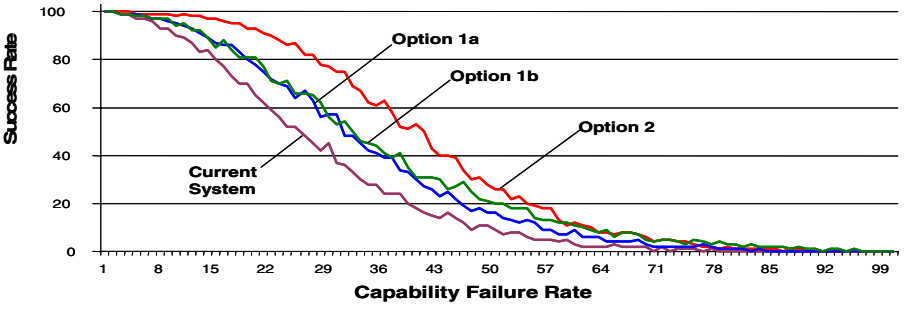


Fig. 6. Comparison of Possible System Updates

Table 6. Additional Agent System Designs Based on Agent Capabilities

Name	AS5	AS6
A1	org, search, move	org, search, move
A2	search, move, vacuum	search, move, vacuum
A3	vacuum, sweep	vacuum, sweep
A4	sweep, mop	sweep, mop
A5	org, mop	org, mop
A6	search, move, sweep	search, move, vacuum

To validate the metric results, we extended our simulation to include the definition of AS5 and AS6. The results of the four different options are shown in Fig. 6 where it is obvious that option 2 provides the best results followed by option 1a and 1b, which are very close. However, all three options are significantly better than the original system, which are consistent with the metric results that we obtained from Bogor.

7 Related Work

Software metrics as a subject area has been around for over 30 years. A number of metrics have been developed to predict or measure various parameters of software systems for different stages of software development lifecycle. For example, metrics to predict software performance were studied in [19, 20], software scalability in [20, 21], software adaptability in [17]. However, metrics and measures for intelligent software systems are as yet vaguely defined and sometimes controversial [2] and are not used extensively in software engineering [10]. There is also little work done in designing and applying metrics at the design level to predict adaptive systems per-

formance. We are proposing new design metrics and examine in details one such metric for distributed, adaptive systems in this paper.

Fault Tree Analysis (FTA) has been studied and used extensively [198]; it provides a top-down approach to systematically describe the combinations of possible occurrences in a system that results in undesirable outcomes such as failures or a malfunctions. Our approach complements FTA, since our technique *automatically* predicts the system flexibility for a given system configuration i.e., it generates traces of system behaviors. If failure patterns are exposed in the traces (e.g., an agent is not used after a certain time), then FTA can analyze the possible set of failure points.

8 Conclusions and Future Work

While the work presented in this paper is a starting point (i.e., there are many additional metrics that must be considered for providing a thorough design evaluation), there are several conclusions that can be made. Likewise, there are several assertions we can make about future work in the areas of performance prediction, additional metrics, scalability, and the integration of metric computations into a model design tool.

Performance Prediction: From the results presented in the previous section, it seems clear that the coarse redundancy rate does predict the flexibility of the robot systems. Unfortunately, system design is seldom as simple as maximizing one metric or parameter. Increased flexibility increases the number of possible assignments that can be made and thus increases the computation burden of generating near optimal assignments at run time. Obviously, a tradeoff exists. In future work, we hope to define additional predictive metrics that a designer can use to help tune the system at design time by performing tradeoff analysis. Our research will not eliminate this predicament, but give the designer predictive numbers to use in making those tradeoffs without developing expensive prototypes/simulations.

Additional Metrics and Query Environment: Based on the state-space analysis in Section 4, we believe the following metrics are helpful; however, we are still working on simulation methods to validate them:

- *Relative Cost Efficiency* (RCE): Using the *potential* function described in Section 3, we can determine path potentials in a goal-role-agent achievement state-space (**GRA**). This would be useful in defining a relative measure of the most/least efficient assignments and giving designers a feel for the organization's best/worst performance. (The actual best/worst performance of the system is not necessarily interesting as either all the agents may fail or the organization's goal may be achieved by changes in the environment.) Thus, the RCE metric would give reasonable feedback about organization instances. If the *potential* function always returns a constant value, thus, it reduces the metric to the shortest/longest achievement paths.
- *Relatively Optimistic Time Efficiency* (ROTE): This metric gives us the most optimistic best/worst time (logical ticks) to achieve the top goal. Consider the path A:{}→ B:{1}→ C:{1, 2(C)}→ D:{1, 2(T), 2(C)}→ E:{1, 2(T), 2(C), 3.1.1(T)}→ F:{1, 2(T), 2(C), 3.1.1(T), 3.2(C)}→ G:{1, 2(T), 2(C), 3.1.1(T), 3.1.2(T), 3.2(C)}. Note that optimistically, 2(C) and 2(T) can be achieved at the same time because

there are no precedence relation among them; similarly with 3.1.1(T) and 3.2 (C). Thus, if we group goal achievements that can happen at the same time, we have the sequence: {A}→{B}→{C, D}→{E, F}→{G}, i.e., 5 logical time ticks. This sub-path grouping approach is akin to partial order reduction techniques in model checking [6] where independent transitions can occur at the same time, thus, one ordering representative of the sub-path is enough. In our case, goal dependences are determined based on the precedence relation. Note that this grouping can also be done in the goal-role-agent achievement paths. If an agent cannot achieve goals simultaneously, the algorithm does not group goal achievements by the same agent in one group. Thus, system designers can evaluate different goal structures, role models, and agent systems for time efficiency. We plan to investigate using these dependence relations for partial order reduction in the near future.

We believe that there are more metrics that can be mined from the state-spaces of system designs. In addition, we also believe that work on query languages (e.g., [13]) can be used to ease system designers when evaluating multiagent system designs.

Extrapolation Methods for Scalability: Note that we do not actually need to use all five agents of the same type (i.e., agents with like capabilities) when exploring the state-space, for example, for AS0; it is sufficient to use one agent for each type (i.e., symmetry reduction [12]), and then extrapolate the actual number of paths based on the paths using representative agents. For example, if we use only one agent for AS0, the number of paths in $\mathcal{GRA}$ is 10. For each path, there are six goals achievements, thus, if we extrapolate each path when using five actual agents, we will have $10\times5^6=156250$ actual paths (which is the one we have from Bogor when directly using 5 agents). Thus, we believe that we can apply symmetry reduction on agent instances based on their type (i.e., they are indistinguishable at the design level) along with the partial order reduction technique hinted above, and use extrapolation methods to recover the actual paths for further analysis.

Integration of Metric Computations in a Model Design Environment: While we manually generated the Bogor configurations for this paper, it would be straightforward to automate such analysis by integrating Bogor into a multiagent design tool. We are currently developing agentTool III (aT^3), an advanced version of the agentTool system for developing organization-based multiagent systems [1]. aT^3 is being developed as an Eclipse plug-in and Bogor already works within the Eclipse plug-in environment. In the integrated system, designers will graphically create system goal, role, and agent models in aT^3 and will simply "click" on a button to popup an interface to select various analysis options; aT^3 will then automatically generate the appropriate configuration and invoke Bogor to explore its state-space and to predict its flexibility.

References

1. agentTool Website: http://macr.cis.ksu.edu/projects/agentTool/agentool.htm
2. Albus, J. S. Metrics and Performance Measures for Intelligent Unmanned Ground Vehicles. Proceedings of the 2002 Performance Metrics for Intelligent Systems Workshop, 2002.

3. Blau, P.M. & Scott, W.R., Formal Organizations, Chandler, San Fran., CA, 1962, 194-221.
4. Bogor Website: http://bogor.projects.cis.ksu.edu
5. Bosnacki, D., Dams, D., Holenderski, L. Symmetric Spin. Proceedings of the Seventh International SPIN Workshop. LNCS 1885, pp. 1-19, 2000.
6. Clarke, E., Grumberg, O., Peled, D. Model Checking. MIT Press, 2000.
7. DeLoach, S.A., & Matson, E. An Organizational Model for Designing Adaptive Multi-agent Systems. The AAAI-04 Workshop on Agent Organizations: Theory and Practice. 2004.
8. Dwyer, M.B., Hatcliff, J., Robby, Prasad, V.R. Exploiting Object Escape and Locking Information in Partial Order Reduction for Concurrent Object-Oriented Programs. International Journal of Formal Methods in System Design (FMSD), 25(2/3), pp. 199-240, 2004.
9. Ericson, C. Fault Tree Analysis – A History. Proceedings of the 17th International System Safety Conference – 1999.
10. Fenton, N.E., Neil, M. Software metrics: roadmap. ICSE-Future of SE, pp 357-370, 2000.
11. Holzmann, G. J. State Compression in SPIN: Recursive Indexing and Compression Training Runs. Proceedings of the Third International SPIN Workshop, 1997.
12. Ip, C. N., Dill, D. L. Better Verification Through Symmetry. International Journal of Formal Methods in System Design (FMSD), 9 (1/2), pp. 47-75, 1996.
13. Liu, Y. A., Stoller, S. D. Querying Complex Graphs. Proceedings of the Eighth Intl Symposium on Practical Aspects of Declarative Languages (PADL). Springer-Verlag, 2006. (to appear)
14. Matson, E., DeLoach, S. Capability in Organization Based Multi-agent Systems, Proceedings of the Intelligent and Computer Systems (IS '03) Conference, 2003.
15. Robby, Dwyer, M.B., Hatcliff, J. Bogor: An Extensible and Highly-Modular Model Checking Framework. Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003), pp. 267-276, 2003.
16. Robby, Dwyer, M.B., Hatcliff, J., Iosif, R. Space-Reduction Strategies for Model Checking Dynamic Software. Proceedings of the 2nd Workshop on Software Model Checking (SoftMC 2003). Electronic Notes in Theoretical Computer Science, 89 (3), Elsevier, 2003.
17. Subramanian, N., Chung, L., Metrics for Software Adaptability, Applied Technology Division, Anritsu Company, Richardson, TX, USA, 2000.
18. van Lamsweerde, A., Darimont, R., Letier, E. Managing conflicts in goal-driven requirements engineering. IEEE Transactions on Software Engineering. 24(11), pp 908-926, 1998.
19. Verkamo, A.I., Gustafsson, J., Nenonen, L., Paakki, J. Design patterns in performance prediction. Proceedings of the Second International Workshop on Software and Performance, ACM Press, pp 143-144, September 2000.
20. Weyuker, E. J., Avritzer, A. A metric for predicting the performance of an application under a growing workload. IBM Systems Journal, Vol 41, No 1, pp. 45-54, 2002.
21. Weyuker, E. J., Avritzer, A. A Metric to Predict Software Scalability. Proceedings of the Eight IEEE Symp. on Software Metrics (METRICS 2002), Ottawa, Canada, pp. 152-159, June 2002.

Combining Problem Frames and UML in the Description of Software Requirements

Luigi Lavazza[1,2] and Vieri Del Bianco[2]

[1] Università dell'Insubria, Dipartimento di Informatica e Comunicazione,
Via Mazzini 5, Varese 21100, Italy
[2] CEFRIEL,
Via Fucini 2, Milano 20100, Italy
{lavazza, delbianc}@cefriel.it

Abstract. Problem frames are a sound and convenient approach to requirements modeling. Nevertheless, they are far less popular than other less rigorous approaches. One reason is that they employ a notation that is neither very appealing nor easy to use. The problem frames notation is sufficiently different from other development languages –especially UML– to create an "impedance mismatch": using problem frames to describe requirements does not help the transition to the design phase, makes it difficult for programmers to fully comprehend requirements, and does not favor traceability. As a consequence, problem frames are rarely adopted in software development processes employing UML as a design language. UML itself provides a linguistic support for requirements modeling, which however suffers from several limitations, especially as far as precision and formality are concerned.

The goal of this paper is to combine problem frames and UML in order to both improving the linguistic support for problem frames –while preserving the underlying concepts– and to improve the UML development practice by introducing the problem frames approach, making it seamlessly applicable in the context of the familiar UML language.

1 Introduction

The Problem Frames approach [1] has the potential to dramatically improve the early lifecycle phases of software projects. Problem frames (PFs) drive developers to understand and describe the problem to be solved, which is crucial for a successful development process.

Nevertheless, PF have some limitations that hinder their application in industrial software development processes. In particular, they are not provided with an adequate linguistic support. For instance, by looking at the PF represented in Fig. 1 it is not immediate to see the association of phenomena with domains, or to see which domain controls which phenomena. Moreover, sometimes it is difficult to represent the nature of shared phenomena; e.g., when they involve complex data structures that are much better modeled via classes.

PFs are not equipped with a unique and clear way for expressing requirements: the modeler has to choose a suitable logic language to predicate about phenomena.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 199 – 213, 2006.

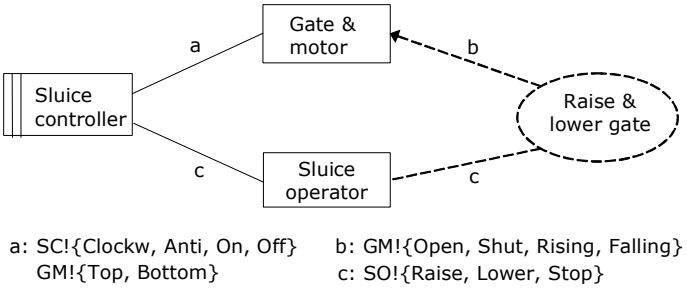


Fig. 1. A commanded behavior frame: the sluice gate control

Moreover, the PF model causes a sort of "impedance mismatch" with respect to the languages employed in the subsequent development phases. In particular, considering object-oriented developments using UML [4] as a modeling language, it is not immediate to transform PF diagrams and predicates into UML models. Even worse, people involved in a UML-based development that are not used to the PF notation can find it difficult to understand requirements. It is well known that design and implementation are more often successful if developers fully understand the requirements; but a Java programmer that works on the basis of a UML design built according to PFs can find it difficult to read the requirements.

Problem frames require a linguistic support that is easy to use, that allows the relevant information to be represented in a natural and readable way, and that allows a smooth transition to the subsequent development phases. In particular, here we consider development activities based on the usage of UML.

For requirements modeling, UML provides the "use case" diagrams [16]. Unfortunately, use cases suffer from several limitations, amply described in the literature [17], [18]. The main limitations with use case diagrams are that they are neither formal nor rigorous–requirements being described mainly by the text that illustrates the "courses of action"– and they are not intrinsically object-oriented, thus it is not easy to move from use cases to the object-oriented models required by the following phases of development. UML-based development needs to be supported by a technique for requirements modeling that overcomes the limitations of use cases.

We propose to integrate PF and UML: PFs are represented by means of UML, so that they can be used (*also*) in UML-based development. This integration provides several benefits: on one hand it equips PFs with a popular and easy to use notation. Under this respect our contribution is quite similar to other initiatives aiming at providing UML-based access to requirements modeling methods (e.g., KAOS [6]). In fact, a method can be truly successful only if a large number of professionals are sufficiently convinced of its potential to use it in industrial settings. Using UML to support requirements engineering with PF may help achieve this end.

On the other hand, the usage of UML both as the notation underlying PFs and as a design language smoothes the transition from the requirement elicitation and modeling phase to the design phase –thus facilitating the comprehension of requirements by the developers– and it eases the iteration between problem and solution domains. Moreover, it makes easier to represent traceability relations, since requirements and elements of the solution are represented in a homogeneous way.

The paper is organized as follows: Section 2 illustrates how problem frames can be defined using UML-based notations. Section 3 tests the ability of the proposed approach to deal with the representation of a couple of concerns that often occur in requirements. Section 4 briefly accounts for related work, while Section 5 draws some conclusions.

2 Problem Frames with UML

The most obvious choice for representing requirements in a way that eases UML-based development is to employ class diagrams (and possibly object diagrams) to represent the structure of the system, and statecharts and OCL statements to specify the behavior of the system [5]. The recently released UML 2.0 [4] features an improved ability to model the structure of systems. In fact, it introduces "composite structures", that allow the modeler to hierarchically decompose a class into an internal structure. The relations among the parts can be specified by means of associations and interfaces.

Unfortunately, OCL [19] is limited with respect to the possibility of specifying temporal aspects: only invariant properties can be formalized, which at most include references to attribute values before or after method execution. It is not possible to reference different time instants in a single OCL formula; namely it is not possible to reference the time distance between events. Therefore, several kinds of important temporal properties of systems cannot be adequately specified.

In order to overcome the aforementioned limitations, Object Temporal Logic (OTL) was defined as a temporal logic extension to OCL [3]. However, OTL extensions do not require to change the OCL metamodel: they can be considered the minimum enhancements of OCL required to deal with time.

OTL formulas are evaluated with respect to an implicit current time instant. In fact, OTL introduces a new primitive as a method of class `Time`: method `eval` receives an `OclExpression` as the parameter (`p`) and returns the (boolean) value of `p` at time `t`. This is denoted as `t.eval(p)` or, more concisely, as `p@t`. All other typical temporal operators –like `Always`, `Sometimes`, `Until`, etc.– are defined based on method `eval`. In addition, OTL allows the modeler to reason about time in a quantitative fashion. Properties can be expressed on (possibly infinite) collections of objects of class `Time`, i.e., on time intervals.

In the rest of the paper OTL is used to express time-dependent properties. Of course, a modeler could express properties and requirements informally, by means of comments associated with model elements. This practice –which is in line with the way UML is generally used in industry– is often acceptable. Nevertheless, here we employ OTL extensively, in order to show that the proposed approach can lead to very precise and rigorous specifications.

2.1 Problem Frames with UML and OTL

Problem frames are represented by means of UML diagrams and OTL statements according to the following rules.

Domains (including the machine) are represented by means of components. This seems a reasonable choice, since components can include any of the properties that characterize domains. In particular it is possible to structure domains into subdomains, while the machine is treated as a sort of "black box component", whose internal details are not given. Different kinds of domains can be represented by means of different kinds of components: we use UML stereotypes to specialize components into causal domains, biddable domains, machines, etc.

Components hide their internal details from the outer world by means of ports. Shared phenomena are represented by means of interfaces, the construct provided by UML to explicitly exchange information among components. In particular, phenomena that are shared between domains D1 and D2 require that components representing D1 and D2 are suitably connected, i.e., they must be equipped with ports and compatible interfaces. Shared phenomena define the interfaces, i.e., they define the operations belonging to the interfaces. Parameters of phenomena become parameters of the operations associated with the interfaces. The details of the domains can be defined in terms of classes or sub-components.

For instance, the required behavior frame (Fig. 2) can be represented by the UML model illustrated by Fig. 3.

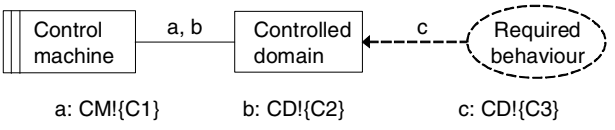


Fig. 2. The required behavior frame

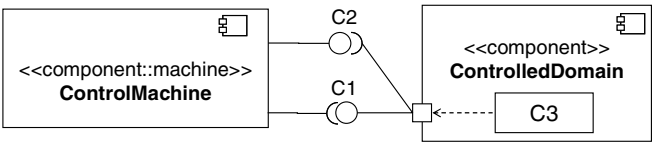


Fig. 3. UML representation of the required behavior frame

Properties are simply expressed as OTL statements involving the proper interface operations and component properties (attributes, states, etc.). Properties of the controlled domain represented in Fig. 3 would be described by OTL statements concerning the context of the `ControlledDomain`. Alternatively, the behavior of the controlled domain can be expressed by means of statecharts. The requirements for the system are expressed similarly by OTL statements concerning the context of `ControlledDomain`: OTL can be used to describe the required behavior of `C3`.

2.2 A Commanded Behavior Frame: The Sluice Gate Control

In order to test the applicability of the proposed approach, in this section we represent the sluice gate control problem by means of UML and OTL.

The problem is defined as follows [1]. A small sluice, with a rising and a falling gate, is used in a simple irrigation system. A computer system is needed to raise and

lower the sluice gate in response to the commands of an operator. The gate is opened and closed by rotating vertical screws. The screws are driven by a small motor, which can be controlled by clockwise, anticlockwise, on and off pulses. There are sensors at the top and bottom of the gate travel; at the top it is fully open, at the bottom it is fully shut. The connection to the computer consists of four pulse lines for motor control, two status lines for gate sensors, and a status line for each class of operator command. The PF diagram for the sluice gate problem is reported in Fig. 1.

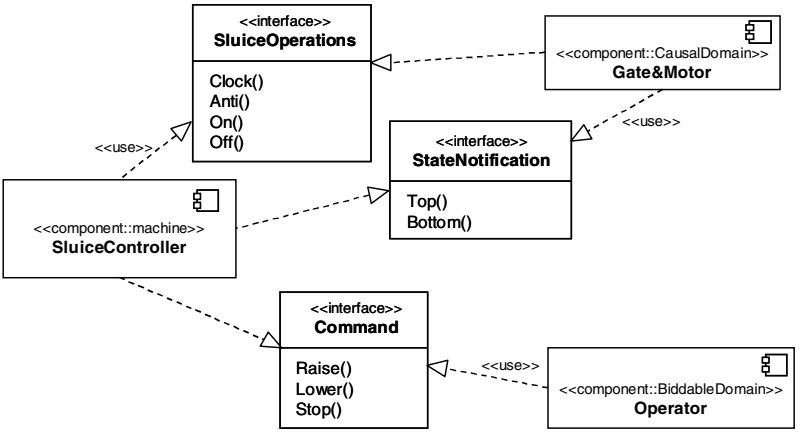


Fig. 4. Sluice gate control: component diagram

The sluice control problem is represented according to our proposal by the class diagram reported in Fig. 4 and by the component diagram reported in Fig. 5. Note that for the sake of clarity, instead of using the "lollypop" notation, in Fig. 4 and Fig. 5 we described the interfaces explicitly, reporting the operations that can be invoked through every interface. It can be noticed that the diagrams incorporate a few choices that improve the clarity of the representation:

- The `Gate&Motor` domain is decomposed into the `Gate` and `Motor` subdomains.
- The events from the `Gate&Motor` are divided into two sets. The messages concerning the state of the gate are emitted by the `Gate` subdomain, while those concerning the control of the motor are delivered to the `Motor` subdomain.
- The `Gate` and `Motor` subdomains are connected via the `GateMotor` interface, in order to represent the effect of the motor on the gate.

The behavior of the controlled domain can be described by the OTL statements reported below. Note that for space reasons we do not give the complete specification, but only a subset that is representative of the capabilities of the proposed approach.

The specifications reported below apply only when the gate and motor are working correctly. If it were required to handle failures, then the combined behavior of the two subdomains would be modeled differently.

- The motor is on, if and only iff an `On` command arrived, and since then no `Off` command arrived:

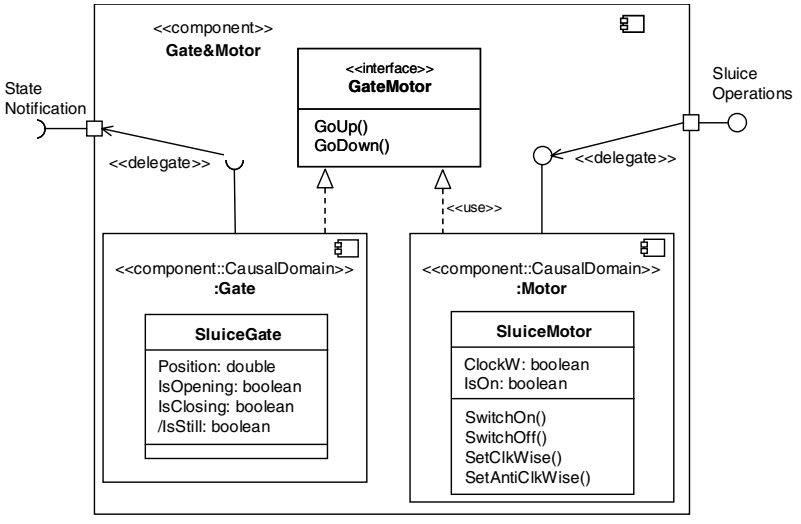


Fig. 5. Sluice gate control: white-box view of the Gate&Motor component

```
context Gate&Motor inv:
SluiceMotor.IsOn = Since(not SluiceOperations^Off,
SluiceOperations^On)
```

`Since(p,q)` states that `q` occurred in the past, and since then `p` is true.

- The gate is still whenever it is neither opening nor closing:

```
context SluiceGate inv:
  self.IsStill= not(self.IsOpeneing or self.IsClosing)
```

- When the motor is on and moving clockwise the gate is opening:

```
context Gate&Motor inv:
  (SluiceMotor.IsOn and SluiceMotor.ClockW) =
   SluiceGate.IsOpening
```

- When the motor is off the gate is still:

```
context Gate&Motor inv:
  SluiceMotor.IsOff = SluiceGate.IsStill
```

- When the gate is opening and the position of the gate reaches the top, the `Open` notification is sent via `StateNotification` interface port within `RT` time units, `RT` being the allowed reaction time.

```
 context Gate&Motor inv:
 (SluiceGate.IsOpening and becomes(SluiceGate.position=top))
  = WithinF(StateNotification^Open, RT)
```

The meaning of `WithinF(p,d)` is that `p` will hold within `d` time units after the evaluation time.

- When the gate is opening the position increases at a constant speed (and similarly the position decreases when the gate is closing):

```
context SluiceGate inv:
 Lasted(self.IsOpening, D) =
 self.position = self.position@(now-D) + D*speed
```

The meaning of `Lasted(p,d)` is that `p` held for `d` time units in the past.

– When the gate is still the position is constant:
```
context SluiceGate inv:
 Lasted(self.IsStill,D)=(self.position=self.position@(now-D))
```
– The correspondence between operations of the interfaces and methods provided by classes or subcomponents can also be specified by means of OTL. For instance, the following statements specify that the `Clock` and `On` commands that reach the `Gate&Motor` domain are delegated to the `SetClkWise` and `SwitchOn` methods of `SluiceMotor` respectively.
```
context Gate&Motor inv:
  SluiceOperations^Clock = SluiceMotor^SetClkWise
context Gate&Motor inv:
  SluiceOperations^On = SluiceMotor^SwitchOn
```
Note that by the `Element^Message` notation we denote both the events of sending and receiving a message.
– Additional trivial statements –not reported here– are needed to enumerate the possible states for the gate and motor, to enforce mutual exclusion of states, to specify the initial states of the gate and motor, etc.

It is interesting to note that the behavior of the combination of the given subdomains was modeled quite naturally by establishing correspondences between the states of the gate and the motor, since the gate is affected by the state of the motor (e.g., when the motor is on clockwise the gate is opening). Otherwise, modeling the behavior of the gate in terms of the events that are directed to the motor would not have been as easy, since the gate is not directly affected by the signals that are sent to the motor. In these situations, the availability of OTL eases the description of the continuous behavior of a domain. Specifying that a domain continuously affects another domain would be very hard using only statecharts.

The behavior of the `Operator` is not specified, i.e., he/she could generate any sequence of commands. It is the job of the sluice controller to guarantee a reasonable behavior for any possible sequence of commands. Here we assume that the requirements for the sluice gate controller are quite simple:

– The `Raise` and `Lower` commands from the operator are simply transformed by the machine in pairs of commands (<`ClockWise, On`> and <`AnticlockWise, On`> respectively), which are sent to the `Motor`, unless the `Gate` is not already `Open` or `Shut`, respectively. This behavior is modeled (for the `Raise` command) by the following OTL code, where `MD` is the time taken by the motor to react to commands, and `ST` is a very short time.
```
context Operator inv:
 (Command^Raise and not SluiceGate.position=Top) =
 (SluiceOperations^Clock and SluiceOperations^On)
context Gate&Motor inv:
 (SluiceMotor^SwitchOn and
  WithinP(SluiceMotor^SetClkWise, ST) or
  SluiceMotor^SetClkWise and
  WithinP(SluiceMotor^SwitchOn, ST))
 and not WithinP(SluiceMotor^SetAntiClkWise, ST)
 and not WithinP(SluiceMotor^SwitchOff, ST))
 implies WithinF(SluiceMotor.IsOn and SluiceMotor.ClockW,MD)
```

Note that last statement takes into consideration the possibility that `SwitchOn` and `SetClkWise` do not arrive at the same time, but (quite realistically) separated by a small interval (≤ `ST`). The statement specifies that if the `On` and `SetClkWise` commands arrive (in any order) in a time interval `ST` and no counter order arrives in the same interval, then the motor will be on in clockwise direction within `MD` time units.

– The `Top` and `Bottom` notifications from the gate, as well as the `Stop` command from the operator are transformed by the machine into the command `Off`, in order to stop the `Motor`.

```
context SluiceController inv:
  (Command^Stop or StateNotification^Bottom or
   StateNotification^Top) = SluiceOperations^Off
context Gate&Motor inv:
   SluiceOperations^Off = SluiceMotor^SwitchOff
```

As already mentioned, we could express more complex requirements. For instance, the controller, instead of just "translating" the commands from the operator, as specified above, could ignore any command that is not separated from the previous one by a minimum interval `D`. This requirement for command `Raise` can be expressed as follows:

```
context Operator inv:
 let AnyOpCommand: OclMessage =
     (Command^Raise or Command^Lower or Command^Stop) in
 (Command^Raise and not SluiceGate.position>=Top and
  not WithinP(D, AnyOpCommand) =
 (SluiceOperations^Clock and SluiceOperations^On)
```

3 Other Issues

In order to further test the applicability of the approach, in this section we consider two of the concerns described in [1]. The concerns are illustrated by means of the same examples reported in [1].

3.1 The Reliability Concern

Several real-life problems can be more easily understood and described if they are decomposed into subproblems. For example, it is often advisable to deal with the reliability concern in a separate subproblem. Subproblems are best identified as projections of the original problems, i.e., they are obtained by considering the subsets of elements and phenomena that are relevant to the subproblem [1].

Reliability is one of the most common subproblems, since often it is necessary to describe the main problem, and then to take into account reliability issues. In the case of the sluice gate controller, it is quite clear that the specifications given in Section 2.2 apply only when the gate and motor work correctly. However, in general it would be desirable that the sluice gate controller guarantees a reasonable behavior of the motor and gate system even in presence of failures. For instance, if debris are jammed in the gate, preventing it from closing, or if a sensor sticks on or off, the controller should not insist in trying to close the gate.

A way to tackle the problem –proposed in [1]– consists in observing the system, in order to provide the operator with a stop warning whenever there is the perception that something is going wrong. This "auditing" subproblem is described in Fig. 6, where the auditing machine is in charge of monitoring the system, represented by the `Gate&motor+Controller` domain, which includes both the controlling machine (the `Sluice controller` of Fig. 1) and the controlled domain (the `Gate&motor` of Fig. 1). The `Audit machine` receives all the relevant events generated by the union of the `Gate&motor` and the `Sluice controller` and reacts sending a `Stop` warning to the safety operator when appropriate. The `Safety operator` (who may be the same person as the `Sluice operator`) is a biddable domain: he/she should be instructed to issue a `Stop` command whenever a `Stop` warning is received.

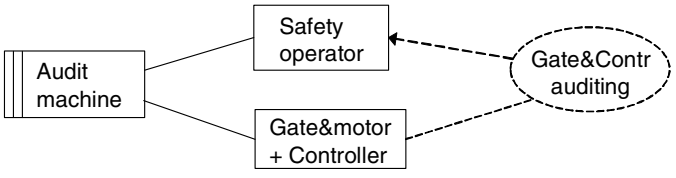


Fig. 6. Sluice gate controller: auditing reliability

Fig. 6 is a projection of the whole sluice gate control problem, and it is an *information display* frame. It is therefore interesting to analyse if the UML-based representation of PFs can manage the representation of a projection of the problem, in the form of an information display frame. It is also interesting to consider how the two resulting sets of UML diagrams can be composed.

The required projection of the system is represented in Fig. 7. The collaboration diagram of Fig. 7 is a rather straightforward representation of Fig. 6. The `Alarm` interface between the `AuditMachine` and the `SafetyOperator` consists just of the warning issued by the machine, to which the operator should react by means of a `Stop` command. Between the `AuditMachine` and the `Gate&Motor&Controller` the `Command&State` interface notifies the `AuditMachine` of the `ClockW`, `Anti`, `On`, `Off` commands and `Top` and `Bottom` states, so that the machine can detect abnormal behaviors of the gate and motor in reaction to commands. The refinement of the diagram reported in Fig. 7 is omitted for simplicity.

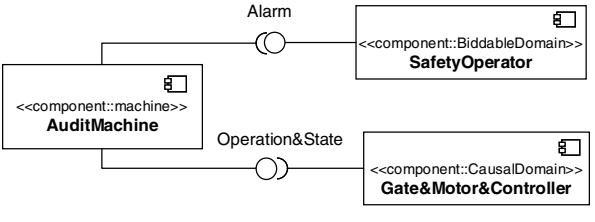


Fig. 7. Sluice gate auditing: component diagram

The requirements for the auditing system can vary widely, depending on the failures considered. Here we consider a failure detection technique based on the assumption that a failure occurs when the gate fails to react to a command in a given interval.

For instance, the following OTL statement specifies the detection of a failure concerning the lowering of the gate. If the condition for starting lowering the gate (i.e., the motor received the <AnticlockWise,On> commands) was verified D time units ago, and D is big enough to allow the completion of the operation (CT being the expected completion time and MD the motor reaction time), and no counter-order was received, and the gate sensor did not notify the completion of the operation (Bottom signal) then a StopWarning will be issued within one time unit. In the following OTL statement the interface names reported in Fig. 8 are used.

```
context AuditMachine inv:
  (Operation^On and Operation^Anticlockwise)@now-D
  and not WithinP((Operation^Clockwise or Operation^Off),D)
  and D >= CT+MD and not WithinP(State^Bottom, D)
 implies WithinF(Alarm^StopWarning, 1)
```

The OTL specifications should be carefully studied in order to compose correctly. For instance, the statement above says that under some conditions the stop warning is issued, but it does not exclude that the stop warning is issued under other circumstances too, or even for no reason at all. The latter case should of course be excluded.

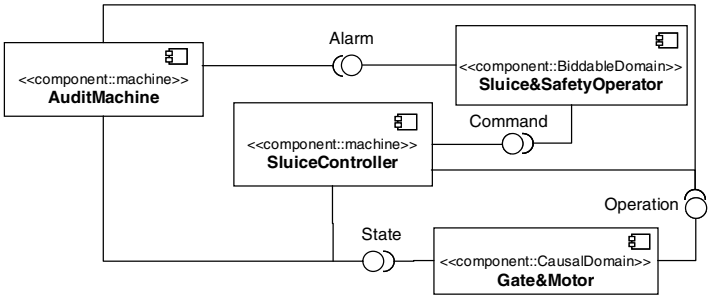


Fig. 8. Sluice gate auditing: component diagram with the Gate&Motor&Controller domain decomposed into its subdomains

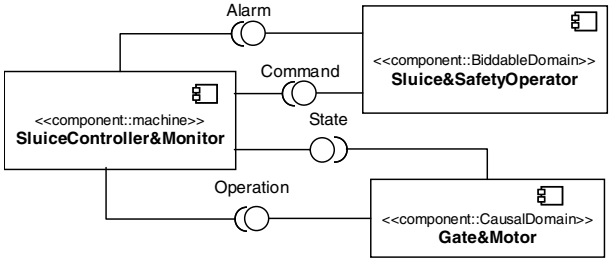


Fig. 9. Sluice gate control and auditing: the composite component diagram

Fig. 8 represents the same diagram as Fig. 7, but with the `Gate&Motor&Controller` domain explicitly decomposed into its sub-components, which belong to the "main" PF reported in Fig. 4 and Fig. 5. It is now possible to note that the diagram in Fig. 8 includes two machines that share the same input and output signals: this suggests that the two machines can actually be merged into a single one, as is normal (and actually required in PF diagrams). The composition of the two subproblems concerning the sluice gate is reported in Fig. 9: this is the complete system that takes into account both functional and reliability requirements.

3.2 The Identity Concern

An identity concern arises when the machine has an interface of shared phenomena with individuals in a multiplex domain. A multiplex domain consists of multiple instances of a class that are not connected into any structure that identifies them, and that do not identify themselves [1].

This concern exists in the following problem. A patient monitoring program is required in a hospital. Each patient is monitored by an analog device which measures factors such as pulse, temperature, blood pressure, and skin resistance. The program reads these factors on a periodic basis (specified for each patient) and stores the factors in a database. For each patient, safe ranges for each factor are also specified by medical staff. If a factor falls outside a patient's safe range, or if an analog device fails, the nurses' station is notified. A simplified version of the problem diagram for the system is reported in Fig. 10.

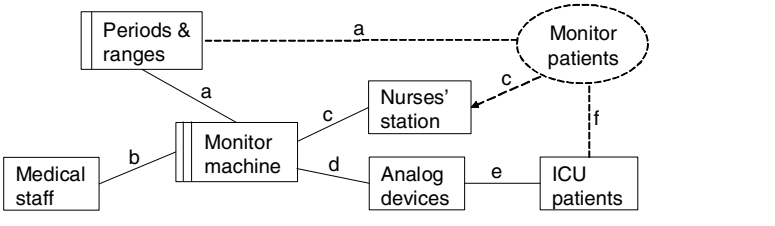


Fig. 10. Patient monitoring: partial problem diagram

Here the identity concern stems from the fact that periods and ranges are associated with patients' names, while the machine gets values that are referred to the devices (more precisely, the device and the machine share the value of a register, accessed at a machine port or storage address). It is therefore necessary to associate each patient name with the correct set of devices. This is done via an identities model domain.

The creation and maintenance of the identities model is a separate subproblem from its use. Such subproblem can be modeled as the *workpieces* PF reported in Fig. 11, where the modeled reality is explicitly represented.

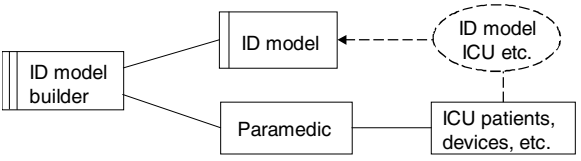


Fig. 11. Identities model creation: problem diagram

For space reasons we do not report here the whole UML model of the patient monitoring system. Instead, we illustrate the parts that are more relevant with respect to the identities concern. Fig. 12 reports the collaboration diagram of the identity modeling subproblem. Fig. 13 reports a simplified version of the class diagram of the patient monitoring system: it is possible to see that the `IDmodel` component includes a set of `Mapping` instances, each one representing a triple <`Patient`, `Device`, `Register`>.

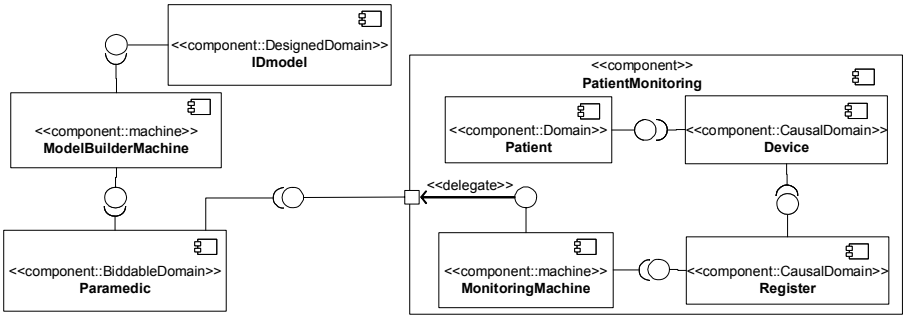


Fig. 12. Identities model collaboration diagram

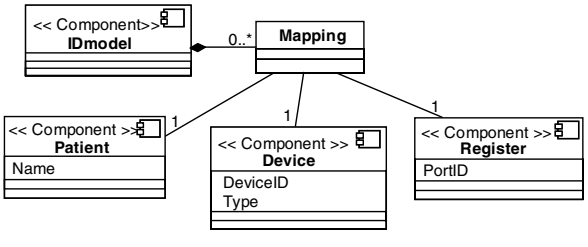


Fig. 13. Identities model: partial class diagram

Because of the simplification of the diagrams above we cannot write syntactically accurate OTL statements. Nevertheless it is easy to see that with OTL we could specify rules like the following: "whenever a patient `P` is attached to a device `D` which is connected to the machine `M` via register `R`, the `IDmodel` must contain a mapping instance associated with `P`, `D` and `R`, and `Rates&Values` for `P` must be available".

Once again, the approach based on the usage of UML and OTL proves adequate to model the considered concern in a quite straightforward way.

4 Related Work

Problem frames have attracted a lot of attention from the researchers. As a result, a relevant amount of work has been done, addressing several aspects of problem frames. A first class of contributions concerns the formalization of problem frames. An early attempt to formally characterize the concepts of problem frames can be found in [12]. In [13] some problem frames are formally specified in the CASL and CASL-LTL specification languages.

A more recent work [14] provides a framework for understanding Problem Frames by locating them within the reference model for requirements engineering [15]. The semantics of problem diagrams is given in terms of "challenges". The semantics supports the textual representation of the diagrams in which Problem Frames capture problems and their relationship to solutions. This work could provide the basis for building graphical tools supporting requirement modeling through Problem Frames.

Since the development process often involves iterative, incremental definition of the problem and solution structures, the need arises to consider architectural structures, services and artifacts as part of the problem domain. Problem frames were extended in this direction, thus permitting an architecture-based approach to software development [2].

A second class of research work considers the usage of PF concepts in conjunction with UML. Konrad and Cheng [11] present a template to describe requirements patterns for embedded systems. Their approach puts together UML and problem frames, without actually merging them properly: they use the PFs to explain the patterns, while UML (class, use case and sequence diagrams) is exploited to illustrate the pattern definition.

Choppy and Reggio proposed UML-based modeling of problem frames [8]. They use plain UML to model the problem domain and the requirements: classes represent domains, while shared phenomena are modeled by means of interfaces. The behavior of the controlled domain is modeled by means of state diagrams and OCL. Requirements are expressed by means of use case diagrams, which are detailed by means of use case descriptions, including statecharts. The modeling approach proposed in [8] is adherent to the UML standard, but is less flexible then ours: because of the limitations of OCL –namely, there is no explicit notion of time– statecharts have to be employed quite extensively. This means that an operational style is enforced, and cases can arise that are difficult to model. The proposal by Choppy and Reggio is actually oriented to the definition of a method for guiding the design phase on the basis of the UML model of the problem frame. For this purpose they provide a family of patterns for the machine design, each presented by a schematic UML model.

5 Conclusions

The goal of the work described in this paper was to experiment with UML-oriented ways of representing problems frames, so that PF-based requirements engineering practices can be effectively integrated into the UML development process.

We showed that problem frames can be actually described by means of UML diagrams complemented with declarative specifications exploiting the OTL language.

A first evaluation of the proposed notation performed on the basis of the examples described in Sections 2 and 3, as well as on additional examples not reported here for space reasons, lets us report the following observations:

- We found no feature of a problem domain, shared phenomenon, behavior specification, etc. that could not be expressed in the proposed UML-based notation. The UML-based notation can be used to document the requirements and specification for an information problem as done with Kovitz's checklists [7]. However, the UML-based notation seems to be more expressive, and to enable a more natural and readable style. For instance, in the sluice control system it is quite natural to represent separately the motor and the gate, to describe the motor in terms of a class with its own properties (attributes and methods), and to map methods onto interface operations, thus contributing to explain the structure and behavior of the controlled domain.
- The UML-based notation can be supported by tools. UML 2.0 compliant tools are currently being released, which support the features of UML that are relevant for our approach. Unfortunately there is no tool support for OTL, however, considering that OTL requires only a small enhancement of the OCL metamodel, it is possible that in the future some OCL tool will be extended to support OTL.
- The UML-based notation favors traceability. With our approach the notation used to describe the problem domain and the requirements is the same used to describe the design. This homogeneity makes it easier to establish/recognize dependency relations, since most relations link elements of the same nature (components, classes, attributes, states, etc.) in requirements and in design. Moreover, several tools for requirements management can import UML models, thus permitting to establish and maintain traceability relationships.
- Describing the requirements with UML makes it possible to define UML-based techniques that guide the transition from the requirements modeling phase to the design phase. An initial experimentation with such technique is reported in [8].

We consider the benefits listed above sufficiently relevant to make the proposed approach appealing in several circumstances. The work presented here can be regarded as a first step towards the definition of a methodology to employ problem frames in a UML-based development process.

References

1. Jackson, M., Problem Frames - analysing and structuring software development problems, Addison-Wesley ACM Press, 2001.
2. Hall, J.G., M. Jackson, R.C. Laney, B. Nuseibeh, and L. Rapanotti, "Relating Software Requirements and Architectures using Problem Frames", Proc. of RE02, Essen, September 2002.
3. Lavazza, L., S. Morasca, and A. Morzenti, "A Dual Language Approach to the Development of Time-Critical Systems with UML" Proc. of TACoS (Int. Workshop on Test and Analysis of Component Based Systems), Barcelona, March 27–28, 2004.
4. OMG, Unified Modeling Language: Superstructure v. 2.0, formal/05-07-04, August 2005.

5. Lavazza, L. "Rigorous Description of Software Requirements with UML", 15th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE2003), San Francisco, July 2003.
6. Heaven, W., and A. Finkelstein. "A UML Profile to Support Requirements Engineering with KAOS". IEE Proceedings – Software , vol 151, no 1, February 2004.
7. Kovitz B., Practical Software Requirements, Manning Publications, 1999.
8. Choppy, C. and Reggio, G., "A UML-Based Method for the Commanded Behavior Frame", 1st International Workshop on Advances and Applications of Problem Frames, co-located with 26th ICSE, Edinburgh, May 2004.
9. Lavazza, L. and Del Bianco, V., "A UML-based Approach for Representing Problem Frames", 1st International Workshop on Advances and Applications of Problem Frame, co-located with 26th ICSE, Edinburgh, May 2004.
10. Selic, B., Gullekson, G., Ward, P. T.: Real-Time Object-Oriented Modeling. Wiley (1994).
11. Konrad, S., Cheng, B., Requirements Patterns for Embebed Systems. IEEE Joint International Conference on Requirements Engineering, Essen, Germany, 2002.
12. Bjørner, D., Koussoube, S., Noussi, R., and Satchok, G. "Michael Jackson's Problem Frames: Towards Methodological Principles of Selecting and Applying Formal Software Development Techniques and Tools". 1st IEEE International Conference on Formal Engineering Methods, Hiroshima, 1997.
13. Choppy, C. and Reggio, G., "Using CASL to Specify the Requirements and the Design: A Problem Specific Approach." In D. Bert and C. Choppy (eds.) Recent Trends in Algebraic Development Techniques, Selected Papers of WADT'99, LNCS 1827, Springer Verlag, 2000.
14. Hall, J., Rapanotti, L., and Jackson, M. "Problem frame semantics for software development", Software and Systems Modeling, vol. 4, n. 2, Springer-Verlag, July 2005.
15. Gunter C.A., Gunter E.L., Jackson M., and Zave P., "A Reference Model for Requirements and Specifications", IEEE Software, vol. 17, n. 3, May-June 2000.
16. Jacobson, I., "Object-Oriented Development In an Industrial Environment", Proc. of OOPSLA '87.
17. Glinz, M., "Problems and Deficiencies of UML as a Requirements Specification Language", Proc. of the 10th IWSPD, San Diego, November 2000.
18. Hurlbut, R.R., "A Survey of Approaches For Describing and Formalizing Use Cases", http://www.iit.edu/~rhurlbut/xpt-tr-97-03.html.
19. OMG, "OCL 2.0 Specification V. 2.0", ptc/2005-06-06, June 2005.

Amplifying the Benefits of Design Patterns: From Specification Through Implementation

Jason O. Hallstrom[1], Neelam Soundarajan[2], and Benjamin Tyler[2]

[1] Department of Computer Science, Clemson University
jasonoh@cs.clemson.edu
[2] Department of Computer Science and Engineering, Ohio State University
{neelam, tyler}@cse.ohio-state.edu

Abstract. The benefits of design patterns are well-established. We argue that these benefits can be further amplified across the system lifecycle. We present two contributions. First, we describe an approach to complementing existing informal pattern descriptions with precise pattern specifications. Our specification language captures the properties common across all applications of a pattern, while accommodating the variation that occurs across those applications. Second, we describe an approach to monitoring a system's runtime behavior to determine whether the appropriate pattern specifications are respected. The monitoring code is generated automatically from the pattern specifications underlying the system's design. We conclude with a discussion of how our contributions are beneficial across the software lifecycle.

1 Introduction

Design patterns [1–3] have become an important part of software practice, fundamentally impacting the design of commercial systems, class libraries, etc. Patterns capture the distilled wisdom of design communities by describing a set of recurring problems, proven solutions to those problems, and the conditions under which the solutions can be applied. They are usually presented as part of a *catalog* that includes a set of patterns relevant to a particular problem domain or application area. When a designer is faced with a design difficulty, the relevant catalogs provide guidance on how to address the difficulty. This idea continues to gain influence; patterns are being discovered and applied in emerging areas as diverse as wireless sensor network design and bioinformatics.

But the benefits of patterns are undercut by three important factors. First, although the informal style used in current pattern catalogs has proven useful, it creates a potential for ambiguity and misunderstanding that jeopardizes the correct use of patterns. This is likely to be a serious problem for team-based projects since different interpretations of a pattern are likely to manifest themselves as incompatibilities among different parts of a system. Second, there is insufficient tool support to assist in discovering pattern implementation errors. Again, these types of tools are especially relevant to team-based projects, where they could be used to detect inconsistent pattern applications. Third, changes

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 214–229, 2006.

introduced during system evolution and maintenance may erode the pattern applications underlying the original design, compromising the *design integrity* of the modified system. The goal of our work is to address these issues, amplifying the benefits of design patterns across the software lifecycle.

We present two contributions. First, we present a *pattern contract language* that captures the structural and behavioral requirements associated with a range of patterns, as well as the system properties that are guaranteed as a result. In addition, the contract language supports *subcontracts*, a form of contract refinement that allows system designers to capture, in a precise way, the customizations made to particular patterns when they are applied. This language will be used to develop *contract catalogs* that complement existing informal pattern catalogs. Second, we present an approach to monitoring a system's runtime behavior to determine whether the system abides by the relevant pattern requirements. The monitoring code for a given system is generated automatically based on the pattern contracts and subcontracts underlying its design.

Before we proceed, it is important to consider a potential problem introduced by developing pattern descriptions that are *precise*. One might argue that existing descriptions are intentionally ambiguous to support flexibility in how patterns may be applied. Precision and flexibility might be at odds here. As we will see, this is not the case. Our approach makes it possible to achieve precision without compromising flexibility. Indeed, in our experience, the process of developing precise descriptions often leads to the discovery of new dimensions of flexibility that are not evident in the informal descriptions.

The rest of the paper is organized as follows. In Section 2, we present a simple pattern-based system, and discuss the difficulties that might be encountered by a software team developing this system. It serves as a running example throughout the paper. In Sections 3 and 4, we present our contract language and contract monitoring approach, respectively. In Section 5, we discuss elements of related work. Finally, in Section 6, we conclude with a summary of our contributions, their benefits to the system lifecycle, and provide pointers to future work.

2 A Pattern-Based Design

To motivate the problems that our work addresses, consider developing a basic simulation of a hospital consisting of doctor, nurse, and patient objects. Each patient is modelled as a quadruple consisting of the patient's name, temperature, heart rate, and a value indicating his/her level of pain medication. Each patient is monitored by a single doctor and multiple nurses that must stay informed of the patient's vital signs. Based on the current readings, doctors and nurses can respond to queries regarding the health of patients under their care. Doctors can also adjust the level of pain medication prescribed to each patient.

The requirement that doctors and nurses stay informed of the current state of their patients calls for the use of the *Observer* pattern. The intent of the pattern is to keep a group of observer objects *consistent* with the state of a subject object. In this case, the observer *role* is played by doctor and nurse objects, and

```
1  public class Patient {
2    private String name; private int temp, hrtRt, medLvl;
3    private Set<Nurse> nurses; private Doctor doctor;
4    ...constructors, field accessor methods...
5    ...addNurse(n), removeNurse(n), setDoctor(d), unsetDoctor()...
6    public void checkVitals() { temp=...; hrtRt=...; notify();  }
7    public void adjustMeds(int newLvl) { medLvl = newLvl; }
8    private void notify() {...call update() on nurses and doctor...} ...
9  public class Nurse {
10   private HashMap<Patient,Integer> vitals; ...constructors...
11   public void update(Patient p) { vitals.put(p, p.getTemp()); }
12   public String getStatus(Patient p) {
13     int t = vitals.get(p);
14     if((t>90)&&(t<105)) return("good"); else return("bad"); } }
```

Fig. 1. Hospital Simulation Code (partial)

the subject role is played by patient objects. Key portions of the *Java* code for this system are shown in Figure 1. When applying the *Observer* pattern, system designers are guided by the pattern description presented in [1]. The style of presentation used in this catalog is common, and consists of an informal description of the problem, a discussion of the properties of the prescribed solution, and UML-like diagrams and code fragments that illustrate canonical applications. This type of description is useful in a number of ways. It is clear from the discussion in [1], for example, that a subject object should provide attach(o) and detach(o) methods for adding and removing an observer (o) from the set of objects *observing* its state. It is also clear that the subject should provide a notify() method that is invoked "whenever a change occurs that could make its observer's state inconsistent with its own." notify() should in turn invoke update() on each attached observer; update() will "reconcile its [the observer's] state with that of the subject." But how will a subject determine whether a change is significant enough to cause it to become inconsistent with its observers? Indeed, what does it mean to say that a subject's state is *inconsistent* with an observer? Similar questions arise when applying other patterns, and are not addressed by the informal descriptions. These are the types of ambiguities that can lead to software defects.

As an example, consider Patient.addNurse(). When this method is invoked, should notify() be called? After all, the execution of addNurse() modifies the state of the patient by adding a new nurse to the patient's set of attached observers. But this modification involves portions of the patient's state that are irrelevant from the point of view of the doctors and nurses already attached to the patient. Stated another way, the change is *insignificant*, and a call to notify() is unnecessary. Consider, however, the attaching nurse. Unless some action is taken, the nurse will not have information about the state of the patient when addNurse() finishes. Hence, if a patient query were issued to this nurse immediately following the

completion of addNurse(), the nurse might return random information about the patient! To prevent this, the addNurse() method must include a call to the update() method of the attaching nurse. This is a subtle issue that is not addressed in the informal description.

Consider a modification to the system. In the new system, nurses are responsible for monitoring vital signs *and* medication levels. The notion of *consistency* will naturally be revised to require that each nurse be aware of the current value of the patient's medLvl field. Designers will presumably revise Nurse.update() to save this information, and also revise Nurse.getStatus() to include the patient's medication level. But this is not sufficient! Changes made to a patient through adjustMeds() will not trigger calls to notify(). As a result, calls to adjustMeds() may leave nurses with inconsistent views of their patients. This is a surprisingly subtle bug given the simplicity of the system. With respect to the pattern, the only change dictated by the new requirements seems to be a redefinition of what it means for the state of a nurse to be *consistent* with the state of a patient — and the corresponding changes in Nurse.update() and Nurse.getStatus(). But as we have seen, this is inaccurate. The change in the notion of *consistency* demands a corresponding change in the notion of *significant change*. More precisely, a change in medLvl is now significant, and should therefore trigger a call to notify(). In general, the *concepts* used in describing a pattern must often satisfy relationships that are not clear from the informal descriptions. Our contracts are designed to make these conditions clear to designers and implementers.

The types of ambiguities that lead to system defects in our hospital simulation are the same types of defects that lead to failures in actual systems. Our pattern specifications are designed to eliminate these ambiguities, while retaining the flexibility present in the informal descriptions. In the event that an implementation error is introduced, our monitoring tools are designed to detect the error before the system is deployed.

3 Design Pattern Contracts

The partial grammar of our contract language is shown in Figure 2. A contract begins with a declaration of the *auxiliary concepts* used throughout its body ($\langle concepts \rangle$). Each specifies a relation involving one or more states of the objects that play *roles* in the pattern being specified. Their purpose is to capture points of variation that occur across different applications of the pattern. Each includes a concept identifier ($\langle coId \rangle$) and the list of roles over which the concept is defined ($\langle rIds \rangle$). The Observer contract, for example, declares the concept *Consistent*(Subject, Observer) to capture the notion of *consistency* between a subject and an observer. Since the meaning of consistency varies from one system built using the *Observer* pattern to another, the contract defers the definition of *Consistent*() to the *subcontract* corresponding to a particular application. By expressing our contracts in terms of auxiliary concepts, but deferring their definitions to subcontracts, we achieve descriptive precision without compromising pattern flexibility. As we saw, however, *arbitrary* flexibility should not be allowed;

```
1  ⟨contract⟩        → pattern contract ⟨pId⟩ {
2     ⟨conceptBlock⟩ ⟨instantiation⟩ ⟨invariant⟩ ⟨roleContracts⟩ }
3  ⟨conceptBlock⟩    → concepts: ⟨concepts⟩ ⟨constraints⟩
4  ⟨concept⟩         → ⟨coId⟩(⟨rIds⟩);
5  ⟨constraints⟩     → constraints: ...predicate on auxiliary concepts...
6  ⟨instantiation⟩   → instantiation: ⟨rId⟩.⟨mId⟩(⟨args⟩) { ⟨cond⟩ };
7                        lead: (target|source|⟨arg⟩|...code...);
8  ⟨invariant⟩       → invariant: ...assertion on roles and concepts...
9  ⟨roleContract⟩    → [lead] role contract ⟨rId⟩ {
10    ⟨fields⟩ ⟨methods⟩ ⟨others⟩ ⟨enrollment⟩ ⟨disenrollment⟩ }
11 ⟨field⟩           → ...role field declaration...
12 ⟨method⟩          → ...standard method specification...
13 ⟨others⟩          → others: ...standard method specification...
14 ⟨enrollment⟩      → ...analogous to instantiation...
15                      enrollee: (target|source|⟨arg⟩|...code...);
16 ⟨disenrollment⟩   → ...analogous to instantiation...
```

Fig. 2. Grammar of Pattern Contracts (partial)

the concept definitions corresponding to a particular system must often satisfy conditions to ensure that the intent of the pattern is not violated. Hence, the pattern contract also specifies *constraints* that must be satisfied by the concept definitions supplied in any subcontract (⟨*constraints*⟩).

The next element specifies the conditions that must be satisfied to *instantiate* a new instance of the pattern (⟨*instantiation*⟩). Pattern instantiation is associated with the invocation of a particular role method or constructor specified in the pattern contract (⟨*rId*⟩.⟨*mId*⟩(⟨*args*⟩)). The contract specifies any state conditions that must be satisfied upon termination of the method, as well as the object that will serve as the *lead object* of the newly created pattern instance. The lead object serves as a handle to refer to its corresponding pattern instance in other portions of the contract. The lead object may be specified as the target of the invocation (target), the source of the invocation (source), one of the arguments to the invocation (⟨*arg*⟩), or some other object specified using a code fragment.

The next element specifies a *pattern invariant* that captures the behavioral guarantees that should be expected if the contract requirements are satisfied (⟨*invariant*⟩). These properties are expressed using an assertion involving the objects enrolled in the pattern instance, and the auxiliary concepts defined by the contract. This assertion will be satisfied whenever control is outside of the participating objects. In effect, this portion of the contract captures the "defined properties" discussed in [2]: the system behaviors that result when the pattern is used correctly.

The final portion consists of *role contracts* that specify the requirements associated with objects *enrolled* to participate in the pattern (⟨*roleContracts*⟩). One of these roles will be flagged as the *lead role*, indicating that its instances may serve as lead objects. The Observer contract, for example, specifies Subject and Observer role contracts, corresponding to the two types of objects that participate in the pattern. The Subject role is flagged as the lead role. Each role contract

begins by specifying the *role fields* to which an object's state must be mapped when it plays the corresponding role (⟨*fields*⟩). Similarly, it specifies the *role methods* that an enrolled object must provide, given the appropriate interface mappings in a subcontract, including pre- and post-condition specifications of the method behaviors (⟨*methods*⟩). These specifications are expressed in terms of role fields and auxiliary concepts. In addition, since an enrolled object may provide methods that do not correspond to any of the role methods, the role contract specifies conditions that prevent these *other* methods from violating the intent of the pattern (⟨*others*⟩). Finally, the role contract specifies the conditions that must be satisfied for an object to *enroll* or *disenroll* (⟨*enrollment*⟩,⟨*disenrollment*⟩). These clauses are defined analogously to the pattern instantiation clause. The only difference in the enrollment clause is that in addition to specifying the lead object (to identify the pattern instance into which the object will enroll), it specifies the enrolling object. The disenrollment clause is analogous.

3.1 Special Notations

Before turning to an example, there are two special notations used in our pattern contracts and subcontracts that are important to consider. The first is the keyword players, used to denote the *sequence* of *player* objects enrolled in a pattern instance. The order of the objects within the sequence corresponds to the order in which the objects enrolled. We use indexing notation to refer to a particular object or subsequence of objects. players[0], for example, refers to the first enrolled object, and players[1:] refers to the subsequence of enrolled objects beginning at the second object.

The second notation allows us to impose conditions on the method calls made by a method during its execution. Addressing such requirements is important since many patterns call for particular methods to be invoked under various conditions. To achieve this, we use the notion of a *call sequence* (or "*trace*"), and use the symbol τ to denote the call sequence associated with a method invocation. Each element within τ represents a method call, and records (*i*) the name of the method invoked, (*ii*) the target of the invocation, and (*iii*) any arguments to the call. We use *dot* notation to denote the projection associated with calls to particular methods of particular objects. τ.o.m, for example, represents the subsequence of calls to method m() on object o. $|\tau|$ denotes the number of calls recorded in the call sequence τ.

3.2 The Observer Contract

Consider the partial contract for the *Observer* pattern shown in Figure 3. The contract declares the auxiliary concepts *Consistent*() and *Modified*(). As explained earlier, *Consistent*() captures the notion of *consistency* between a subject and an observer. *Modified*() captures the notion of *significant change* within a subject. The latter concept is later used to express the requirement that every significant change within a subject result in a call to notify(). The former concept is used to require that Observer.update() appropriately update the observer's state. The constraint imposed on these concepts requires that if a subject's state changes

```
pattern contract Observer {
  concepts:
    Consistent(Subject,Observer); Modified(Subject,Subject);
  constraints: ∀s1,s2,o1:::(¬Modified(s1,s2) ∧ Consistent(s1,o1))
                                  ⇒ Consistent(s2,o1)
  instantiation: Subject.Subject() { obs=∅ }; lead: target;
  invariant: Subject(players[0]) ∧ Observer(players[1:]) ∧ ... ∧
             ∀ob:ob ∈ players[1:]::Consistent(players[0],ob)
```

Fig. 3. Observer Pattern Contract (partial)

from s1 to s2, and the change is deemed *insignificant*, then any observer state consistent with s1 must also be consistent with s2. This constraint prevents the types of incompatible concept definitions that lead to software defects in our hospital system. More precisely, it prevents definitions of *Modified*() and *Consistent*() that would allow a subject to omit a call to notify() after a change that could lead to *inconsistency* with one or more of its observers.

The instantiation clause specifies that a new instance of the pattern is created each time a new subject object is created. Further, it requires that at the point of instantiation, the subject's obs set be empty (since no observers have yet enrolled). Finally, it states that the newly created subject will serve as the *lead* object of the pattern instance.

The invariant clause captures the intent of the pattern, the "defined properties" that may be expected if the contract requirements are met. It states that the first object to enroll in a pattern instance will play the role of Subject and all other enrolled objects will play the role of Observer. Most important, it states that whenever control is outside of the participating objects, all of the enrolled observers will be in states that are consistent with the current state of the subject.

The partial Subject role contract is shown in Figure 4, and specifies the state components and method behaviors that must be provided by objects playing the Subject role. To benefit from the *pattern invariant*, these requirements must

```
lead role contract Subject {
  Set<Observer> obs;
  void attach(Observer ob):
    pre: ob ∉ obs
    post: (ob=#ob) ∧ ¬Modified(#this,this) ∧(obs=(#obs∪{ob}))
          ∧ (|τ|=1) ∧ (|τ.ob.update|=1)   ...detach(ob)...
  void notify():
    post: (obs=#obs) ∧ ¬Modified(#this,this) ∧ (|τ|=|obs|) ∧
          ∀ob:ob ∈ obs::(|τ.ob.update|=1)
  others:
    post: (obs=#obs) ∧ ((¬Modified(#this,this) ∧ (|τ|=0)) ∨
          (|τ.this.notify|=1))
```

Fig. 4. Subject Role Contract (partial)

be satisfied under the field and interface mappings specified in the relevant subcontract. (We will discuss these mappings shortly.) The role contract states that each subject must provide a Set component, which will be used to store the set of attached observers. It also includes specifications for the attach(), detach(), and notify() methods. In the post-conditions of these methods, we use the # notation to denote the pre-conditional value of an object. Hence, the attach() method is required to preserve the reference to the attaching observer (ob), to leave the subject *unmodified*, and to add the attaching object to the set of attached observers (obs). Further, the call sequence conditions require that update() be invoked on the attaching object. This requirement guarantees — given the specification of Observer.update() (omitted) — that the observer will be in a state that is *consistent* with the current state of the subject when attach() terminates. Again, this condition is important to prevent the types of inconsistency defects encountered in our hospital system. detach() is defined analogously, but omits the call sequence conditions. The final method, notify(), is required to preserve the set of attached observers, and to leave the subject *unmodified*. The call sequence conditions require that the method invoke update() on each attached observer.

The others clause imposes requirements on the methods provided by a player beyond those that map to attach(), detach(), and notify(). All of these *other* methods are required to preserve the set of attached observers. Further, if one of these methods makes a *significant change* in the subject (*i.e.*, *Modified*(#this, this) is *true*), it must include a call to notify(). As we have seen, this method will in turn invoke update() on each attached observer, ensuring their *consistency* with the new state of the subject.

The Observer role contract is defined in the same manner as the Subject role contract. observer objects are required to maintain a reference to their subject, and to provide an update() method. The post-condition of update() requires that it leave the observer in a state that is *consistent* with the current state of the observer's subject. The others clause imposes similar requirements to ensure that the pattern invariant is respected.

3.3 Pattern Subcontracts

A subcontract *specializes* a pattern contract for use, customizing its requirements and behavioral guarantees to the needs of a particular system. This specialization mechanism is essential for preserving the *flexibility* of our pattern contracts. The partial grammar of our subcontract language is shown in Figure 5. Each subcontract begins by specifying a set of *role maps* that characterize the manner in which particular system classes can be viewed as their corresponding role types (⟨*roleMaps*⟩). The Hospital subcontract, for example, defines role maps that allow us to view a patient as a subject, a nurse as an observer, and a doctor as an observer. Each role map consists of a *state map* and an *interface map* (⟨*stateMap*⟩,⟨*interfaceMap*⟩). A state map defines functions that map an object's fields to the fields defined by its role (⟨*rfId*⟩). These functions are written in the form of code fragments to simplify the expression of the mappings, and to simplify the task of generating the appropriate monitoring code. Similarly, an

```
1  ⟨subcontract⟩  → subcontract ⟨sId⟩ specializes ⟨pId⟩ {
2    ⟨roleMaps⟩ ⟨concDefBlock⟩ }
3  ⟨roleMap⟩      → rolemap ⟨cId⟩ as ⟨rId⟩ {
4    ⟨stateMap⟩ ⟨interfaceMap⟩ }
5  ⟨stateMap⟩     → state: ⟨fieldMaps⟩
6  ⟨fieldMap⟩     → ⟨rfId⟩ = {...code...}
7  ⟨interfaceMap⟩ → methods: ⟨methodMaps⟩
8  ⟨methodMap⟩    → ⟨rmId⟩(⟨rmArgs⟩):⟨classMethods⟩
9  ⟨classMethod⟩  → ⟨cmId⟩(⟨cmArgs⟩)[{⟨argMaps⟩}]
10 ⟨concDefBlock⟩ → auxiliary concepts: ⟨concDefs⟩
11 ⟨concDef⟩      → ⟨coId⟩(⟨coArgs⟩) {...code...}
```

Fig. 5. Grammar of Pattern Subcontracts (partial)

interface map specifies mappings from the class methods and arguments to their corresponding role methods and arguments (⟨*methodMaps*⟩,⟨*argMaps*⟩). As we will see, multiple class methods may be mapped to a single role method.

The final element of a subcontract provides *auxiliary concept* definitions appropriate to the given system (⟨*concDefs*⟩). Each auxiliary concept is written as a code fragment expressed over the classes mapped to the concept arguments. Each concept returns a boolean value indicating whether the relation is satisfied given the states of the objects passed as argument. When the auxiliary concept definitions and role maps are substituted into the contract being specialized, the resulting specification characterizes the pattern requirements and behavioral guarantees specific to the system in question.

3.4 The Hospital Subcontract

As an example, consider the partial subcontract for our hospital system shown in Figure 6. The subcontract begins by defining the role map that allows us to view a patient as a subject. Under this view, the state map specifies that the subject's obs field is realized as the set containing all of the elements in nurses, plus the object referenced by doc, if any. The interface map specifies that both addNurse(n) and setDoctor(d) play the part of attach(o). In both cases, the argument to the class method corresponds directly to the argument to the role method. removeNurse(n) and unsetDoctor() are defined analogously, except that in the case of unsetDoctor(), which takes no arguments, the argument to detach(ob) is played by the patient's doc field. Patient.notify() corresponds directly to Subject.notify(). The Nurse as Observer, and Doctor as Observer role maps are defined in a similar manner.

The definition of *Modified*() specifies that any change in patient.temp or patient.hrtRt is considered a *significant change*. The two definitions of *Consistent*() are more interesting. Since each nurse and each doctor may be involved in multiple pattern instances, each may store information about multiple patients. Or more generally, each observer may store information about multiple subjects. In

```
1  subcontract Hospital specializes Observer {
2    rolemap Patient as Subject {
3      state: obs = { Set<Observer> obs =
4                       new HashSet<Observer>(nurses);
5                     if(doc!=null) obs.add(doc); return(obs); }
6      methods:
7        attach(Observer ob):addNurse(ob),setDoctor(ob)
8        detach(Observer ob):removeNurse(ob),
9                            unsetDoctor(){ob=doc}   ...notify()...
10   ...Nurse/Doctor as Observer rolemaps...
11   auxiliary concepts:
12     Modified(Patient p1, Patient p2) {
13       return((p1.temp!=p2.temp) || (p1.hrtRt!=p2.hrtRt)); }
14     Consistent(Patient p, Nurse n) {
15       return(p.hrtRt == n.vitals.get(lead)); }
16     ...Consistent(Patient, Doctor) concept definition...
```

Fig. 6. Hospital Subcontract (partial)

reasoning about a particular pattern instance, it must be possible to project out those portions of an observer's state relevant to the pattern instance (and therefore the subject) in question. This is achieved using the lead object (a surrogate pattern instance identifier) as an index into the observer's state. The lead keyword refers to the lead object in a way that is analogous to the use of the this keyword in object-oriented languages. Hence, in the definition of *Consistent*() corresponding to nurse objects, the lead object is used to retrieve the patient information corresponding to the pattern instance in question. The case involving doctor objects is analogous.

4 Pattern Contract Monitors

In addition to having pattern contracts that are both precise and flexible, it is important to have supporting software tools that can assist in determining whether the requirements specified by a contract are satisfied. To achieve this, we have developed a monitor generation tool based on our pattern contract language. Given the pattern contracts and subcontracts underlying a particular system design, our tool generates runtime monitoring code that signals any violations of the contract requirements. Given that the assertions to be checked are crosscutting, we chose to use an *aspect-oriented* approach. Our current implementation targets Java-based systems, and generates aspects in *AspectJ* [4][1]. The monitor generation process is illustrated in Figure 7.

The monitoring code produced for a given contract/subcontract pair consists of one *abstract aspect* and one *concrete subaspect*. The abstract aspect contains

[1] The tool, including source code, documentation, and system examples, is available for download at: *http://www.cse.ohio-state.edu/~tyler/MonGen/*

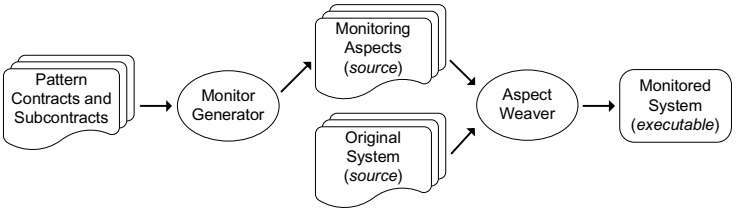


Fig. 7. Monitor Generation Process

checking logic common across all specializations of the contract, and the subaspect tailors this logic to the particular specialization specified by the subcontract. Consider, for example, the abstract aspect generated from the Observer contract (Figures 3 and 4) shown in Figure 8. The aspect begins by declaring interfaces for each of the roles defined in the pattern contract (Line 2). These interfaces are mapped to the appropriate system classes in the subaspect based on the role maps included in the subcontract. The subaspect generated from the Hospital subcontract (omitted), for example, maps the Subject interface to the Patient class using *AspectJ*'s declare parents construct. This effectively forces the Patient class to implement the (empty) Subject interface. Similar mappings are defined for Nurse and Doctor. This allows methods defined in the abstract aspect, which are defined in terms of Subject and Observer objects, to work with Patient, Nurse, and Doctor objects.

The aspect next defines state components required to monitor multiple pattern instances (Lines 3–4). The first of these components is a *pattern instance map* (instanceMap) that maintains a mapping from each lead object to its corresponding PatternInstance object. Each PatternInstance stores information about a single pattern instance, including references to the enrolled objects and the roles that these objects play. This information is required to check certain instantiation, enrollment, and disenrollment conditions — such as those that make use of the players keyword. The instance map is updated when a pattern instance is created or destroyed, and when an object enrolls or disenrolls.

The second state component is the *trace stack* (traces), which stores call sequence (τ) information about each of the active role methods. This information is required to check the call sequence conditions specified in the pattern contract. The trace stack is updated before and after every role method invocation.

The aspect next declares *pointcuts* corresponding to each of the role methods specified in the pattern contract (Lines 5–6). The *advice* bound to these pointcuts is responsible for checking the appropriate role method requirements, as well as for updating the pattern instance map and trace stack. Since, however, the mapping between class methods and role methods varies from application to application, the pointcuts are declared *abstract*. Pointcut definitions are supplied in the subaspect based on the interface maps specified in the relevant subcontract. The subaspect generated from the Hospital subcontract, for example, maps the sub_attach() pointcut (corresponding to Subject.attach()) to the execution of either Patient.addNurse() or Patient.setDoctor(). Similar pointcuts are used to capture pattern instantiation, object enrollment, and disenrollment. Pointcuts

```
1  public abstract privileged aspect ObserverM {
2    interface Subject{} interface Observer{}
3    private HashMap<Subject,PatternInstance> instanceMap;
4    private TraceStack traces;
5    ...pointcuts for role constructors, role methods, and other methods:
6    abstract pointcut sub_attach(Subject _this, Observer ob); ...
7    ...auxiliary concept methods:
8    public abstract boolean Modified(Subject a1, Subject a2);
9    public abstract boolean Consistent(Subject a1, Observer a2);
10   ...role state accessor methods:
11   public abstract Set<Observer> sub_obs(Subject _this);
12   public abstract Subject obs_sub(Observer _this,Subject lead);
13   ...assertion checking / bookkeeping advices:
14   after(Subject _this, Observer ob): sub_attach(_this, ob){
15     ...get #ob, #this from caller trace record...
16     assert((ob==pre_ob) && !Modified(pre_this, _this) &&
17       sub_obs(_this).containsAll(sub_obs(pre_this)) && ... &&
18       (traces.current().length() == 1) &&
19       (traces.current().limit(ob,"update").length() == 1));
20     ...update caller trace record... } ... }
```

Fig. 8. The ObserverM Contract Monitor (partial)

are also declared to capture the *other* methods of the class(es) mapped to each role. These pointcuts are defined to include all of the class methods *except* those bound to role methods.

Recall that the requirements specified in the pattern contract are expressed in terms of auxiliary concepts and role fields. Since the realizations of these elements vary, they are captured using abstract methods, deferring their definitions to a subaspect. ObserverM, for example, declares Modified() and Consistent() methods corresponding to the auxiliary concepts of the same name (Lines 7–9). It also declares abstract methods corresponding to Subject.obs and Observer.sub (Lines 10–12). Each of the latter methods returns the appropriate role field value when the argument passed as input is viewed as an instance of its role. The implementations of the auxiliary concept and role field methods are supplied in the subaspect based on the concept definitions and state maps provided in the relevant subcontract. Since these elements are defined (in the subcontract) in terms of code fragments, the code generation task is straightforward.

Note that for Observer.sub, the corresponding method takes an additional argument. Since the Observer role is not flagged as lead in the pattern contract, each observer may participate in multiple pattern instances. This means that each observer (conceptually) stores multiple copies of the sub field — one copy corresponding to each pattern instance. The lead argument is used to identify the pattern instance under which the state mapping should be performed.

The final portion of the aspect defines the advice bound to each pointcut. The checking code within the advice is generated based on the assertions specified in the pattern contract. The before and after advice bound to each pointcut

	Size of Contract		Size of Subcontract		Execution Time (in ms)	
	Specific.	Abs. Aspect	Specific.	Subaspect	w/ Montr.	w/o Montr.
Observer	1723	14,701	866	3967	2657	172
Memento	907	11,116	771	3730	3610	391
Chain of Resp.	592	5658	377	1845	2453	297

Fig. 9. Code Size and Runtime Overhead. [Pentium-IV @ 2.53GHz, 512MB RAM, Windows XP Pro SP 2, Sun JVM 1.5.0_04].

is responsible for checking the relevant pre- and post-conditions, respectively. The advice is also responsible for updating the pattern instance map and the trace stack. A portion of the after advice generated from the specification of Subject.Attach() is shown in the figure (Lines 14–20). The advice bound to the remaining pointcuts is defined in a similar manner.

One difference between the advice bound to Subject methods and the advice bound to Observer methods is that the latter begins by identifying *every* pattern instance in which an observer participates. The relevant assertions are then checked in the context of each pattern instance. The corresponding lead object is retrieved from the pattern instance map, and serves as the second argument when invoking the role field method corresponding to Observer.sub.

4.1 Code Size and Runtime Overhead

We have applied our approach to several different patterns and systems. Space restrictions preclude a detailed discussion of the results, but it is interesting to consider the gross relationship between contract/subcontract size and the size of the corresponding monitoring code. It is also interesting to consider the runtime overhead introduced when this code is woven into an actual system. Figure 9 presents the data corresponding to the use of our contracts for *Observer*, *Memento*, and *Chain of Responsibility* when used in monitoring the canonical system examples presented in [1]. As a gross estimate, we measure size in terms of non-whitespace characters. We emphasize that this is a preliminary analysis.

5 Related Work

We are not the first to consider pattern formalization. Eden *et al.* [5], for example, propose a higher-order logic formalism that captures patterns as formulae. Each formula consists of a declaration of the participating classes, methods, and inheritance hierarchies, and a conjunctive statement of the relations among them. While rich structural properties can be expressed, there is limited support for capturing behavioral properties. The formalism does not, for example, provide constructs for referring to pre- and post-conditional values, nor does it provide a concept analogous to our method call sequences. By contrast, Mikkonen's work [6] focuses almost exclusively on behavioral properties. In his approach, patterns are specified using an action system notation. Data classes model pattern participants, and guarded actions model their interactions. The approach

is well-suited to reasoning about temporal properties. One limitation, however, is that the separation of actions and data is structurally inconsistent with the OO paradigm, making it difficult to express most structural properties. Further, Mikkonen's specifications cannot be specialized to the needs of particular systems; thus pattern flexibility may be seriously compromised.

Helm *et al.* [7] describe a contract formalism that shares similarities with ours. For example, their formalism includes a construct similar to our auxiliary concepts. It does not, however, provide a way to impose constraints that would prevent definitions of these concepts from violating a pattern's intent. The formalism also includes support for specifying the relative order of method invocations, but the support is limited. It is impossible, for example, to quantify over a method call sequence to require that a particular method be invoked exactly once, or alternatively, that a particular method not be invoked at all. Finally, there is nothing analogous to our use of the others clause to prevent non-role methods from violating a pattern's intent.

In [8] and [9], we describe principles of pattern formalization and runtime monitoring, but do not consider a general pattern specification language, pattern specializations, or automated monitor generation. We provide an overview of the specification and monitoring approach in [10], but do not go into the technical detail presented here. For example, [10] presents only a subset of the specification language; the subset cannot, for example, accommodate multiple pattern instances. Other important contributions presented here that are absent from [10] include a detailed system and subcontract example, a presentation of the generated monitoring code, an analysis of the code size and runtime overhead associated with monitoring, and a discussion of how the approach supports a pattern-centric software lifecycle (Section 6).

Runtime assertion monitoring of OO systems has a long history [11–13], and some authors have considered aspect-based approaches. Lippert and Lopes [14] use *AspectJ* to refactor pre- and post-conditional assertion checking code. Gibbs and Malloy [15] propose using aspects to monitor class invariants involving temporal properties. To our knowledge, however, we are the first to investigate contract monitors for design patterns.

6 Discussion

Our work was motivated by three observations. First, informal pattern descriptions leave a potential for ambiguity and misunderstanding that jeopardizes the correct use of patterns. Second, there is limited tool support to assist in identifying pattern implementation errors. Third, as a system evolves, its *design integrity* may erode under maintenance; it may no longer remain faithful to the patterns underlying its design. We presented two contributions to address these problems. The first was a formalism for expressing *pattern contracts* that capture the implementation requirements and behavioral guarantees associated with a range of patterns. The formalism includes support for *subcontracts* that capture

the ways in which patterns are specialized for use in particular systems. Thus, we are able to specify properties common across all applications of a pattern, while accommodating the inherent variation that occurs across those applications. We illustrated the approach by developing the contract for the *Observer* pattern, and a corresponding subcontract for a simple system built using this pattern.

Our second contribution was a *monitor generation tool*. Given the pattern contracts and subcontracts underlying a system design, our tool produces a set of aspects in *AspectJ* that monitor the system's runtime behavior to check whether the contract requirements are violated. We presented some of the key details concerning the aspects generated by the tool, as well as the structure of the tool itself. Finally, we presented preliminary figures to show the code and runtime overhead involved in using the tool to monitor a system during its execution.

These contributions, along with our planned extensions, provide the basis for a *pattern-centric* software lifecycle. At the foundation of the lifecycle is a *contract catalog* that complements existing pattern catalogs. The catalog is an evolving document that we plan to make accessible through the web. We hope that researchers interested in lightweight formal methods will contribute to its development. Community involvement is essential in ensuring that the contracts faithfully capture the intent of the patterns specified. Members of a design team will be able to consult the catalog to ensure a common understanding of the requirements associated with the patterns underlying a particular design.

As the design and implementation details of the system are fleshed out, part of the design team will be charged with creating the corresponding subcontracts. In addition to guiding the implementation, the subcontracts will allow implementation and maintenance teams to generate appropriate runtime monitoring code. Executing this code will enable the team to identify pattern implementation errors more easily — from early implementation through evolution.

Note that while developing a pattern contract requires reasonable facility with formal notations, developing a subcontract is a task that will likely appeal to system developers. Indeed, this is one reason why this portion of the formalism resembles a programming notation more than it resembles formal mathematics. As part of our future work, we plan to assess the degree of effort involved in developing and maintaining these subcontracts. This will allow us to perform a cost-benefit analysis by comparing this effort to the benefit received when using the approach. We also plan to investigate techniques for generating test suites that ensure suitable *coverage* of the patterns' used in a system.

Another exciting possibility is a *pattern-centric visualization tool*. During a system's execution, the monitoring code will save appropriate information relevant to the patterns used in the system. The visualization tool will then take this information and play it back in the form of a "*slow-motion-video*", allowing the user to go back and forth in the system's execution, focusing on the interactions among groups of objects interacting according to the patterns of interest. This will be of particular value to new members of a design team since it will enable them to quickly develop a pattern-centric understanding of relevant systems.

References

1. Gamma, E., *et al.*: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
2. Buschmann, F., *et al.*: Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons, Inc. (1996)
3. Schmidt, D., *et al.*: Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects. John Wiley & Sons, Inc. (1996)
4. Kiczales, G., *et al.*: An overview of AspectJ. In: ECOOP. (2001) 327–353
5. Eden, A.: Formal specification of object-oriented design. In: CSME-MDE. (2001)
6. Mikkonen, T.: Formalizing design patterns. In: ICSE, IEEE (1998) 115–124
7. Helm, R., *et al.*: Contracts: Specifying behavioral compositions in object-oriented systems. In: OOPSLA/ECOOP, ACM (1990) 169–180
8. Soundarajan, N., Hallstrom, J.: Responsibilities and rewards: Specifying design patterns. In: ICSE. (2004) 666–675
9. Soundarajan, N., *et al.*: Specifying and monitoring design pattern contracts. In: SAVCBS/ICSE. (2004) 87–94
10. Tyler, B., *et al.*: Automated generation of monitors for pattern contracts. In: SAC. (2006) (*to appear*).
11. Meyer, B.: Object-Oriented Software Construction. Prentice Hall (1988)
12. Rosenblum, D.: A practical approach to programming with assertions. IEEE TSE **21** (1995) 19–31
13. Burdy *et al.*, L.: An overview of JML tools and applications. STTT (2005) (to appear).
14. Lippert, M., Lopes, C.: A study on exception detection and handling using aspect-oriented programming. In: ICSE. (2000) 418–427
15. Gibbs, T., Malloy, B.: Weaving aspects into C++ applications for validation of temporal invariants. In: CSMR. (2003) 249–258

The Good, the Bad and the Ugly: Well-Formedness of Live Sequence Charts*

Bernd Westphal and Tobe Toben

Carl von Ossietzky Universität Oldenburg, Oldenburg, Germany
{westphal, toben}@informatik.uni-oldenburg.de

Abstract. The Life Sequence Chart (LSC) language is a conservative extension of the well-known visual formalism of Message Sequence Charts. An LSC specification formally captures requirements on the inter-object behaviour in a system as a set of scenarios. As with many languages, there are LSCs which are syntactically correct but insatisfiable due to internal contradictions. The authors of the original publication on LSCs avoid this problem by restricting their discussion to well-formed LSCs, i.e. LSCs that induce a partial order on their elements.

This abstract definition is of limited help to authors of LSCs as they need guidelines how to write well-formed LSCs and fast procedures that check for the absence of internal contradictions. To this end we provide an exact characterisation of well-formedness of LSCs in terms of concrete syntax as well as in terms of the semantics-giving automata. We give a fast graph-based algorithm to decide well-formedness. Consequently we can confirm that the results on the complexity of a number of LSC problems recently obtained for the subclass of well-formed LSCs actually hold for the set of all LSCs.

1 Introduction

The Live Sequence Chart (LSC) language is a scenario based specification language that is used to formalise requirements on the inter-object behaviour of distributed systems under design. It conservatively extends the well-known Message Sequence Charts [10] basically by introducing modalities. The mode of the whole chart distinguishes example scenarios from scenarios the system must always adhere to, the mode of locations allows to require progress along an instance line, and the mode of conditions, which are semantically meaningful in LSCs, allows to add so called legal exits to a scenario. The modalities make LSCs strictly more powerful than MSCs whereas the graphical appeal and intuitivity of MSCs is preserved by indicating the modalities graphically. Scenario-based approaches in general [16, 1] and LSCs in particular [2, 4, 6, 7, 12] have shown adequate for the specification of requirements on distributed systems.

After a period of experimentation and evaluation, the language has stabilised into the two dialects of [11] and [9] and is subject of active research concerning

* This work was partly supported by the German Research Council (DFG) in SFB/TR 14 AVACS and in project DA 206/7-3 (USE), SPP 1064.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 230–246, 2006.

fundamental properties of the language, e.g. decidability and complexity of problems like realisability [5], and expressive power in terms of temporal logic [13, 15]. The two dialects emerged from two objectives of LSC specification usage. The LSCs of [9] are tailored for the so called play-out approach. They employ a tool called play-engine to execute an LSC specification. Thereby there needn't be an implementation of the intra-object behaviour of the system under design; the set of LSCs *is* the implementation. To this end, they added, e.g., actions to modify the state of the system, loops, and sub-charts to the original proposal. The LSCs of [11] are tailored for a more classical approach, where an LSC specification complements the model-based development of the intra-object behaviour of a system using, e.g., Statemate state-charts or UML state-machines. Whether the intra-object behaviour adheres to the LSC specification can then automatically be established by model-checking as has been demonstrated in [2, 14]. To this end the LSCs of [11] have, e.g., local invariants to state requirements or assumptions on periods of time and activation modes (cf. Sect. 2). We introduce the LSCs of [11] in more detail in Sect. 2 and in the following we mean these LSCs if not otherwise specified. Note that although these two usages of LSCs have been investigated independently, they don't exclude each other at all. Restraining to the common sublanguage of both dialects, one may write (or play-in [9]) an LSC specification, gain confidence into the specification by playing it out, and only then start a model-based implementation of intra-object behaviour that is then formally verified against the LSC specification.

The subject of this paper is an issue that turned up in the many experiments with formal verification for LSCs. There are LSCs that are syntactically correct but that are insatisfiable due to internal contradictions. The most obvious example are instantaneous messages that cross each other like m_2 and m_3 in the LSC body in Fig. 3(a) on page 239. For instantaneous messages, the sending and reception has to be observed simultaneously thus by the order on '$inst_1$' the sending and reception of m_2 has to be observed strictly before m_3. The order on '$inst_2$' requires it the other way round. Thus the LSC shown in Fig. 3(a) is not satisfied by any model. And if it were a pre-chart (cf. Sect. 2), then any model would (trivially) satisfy the whole LSC as the premise of the main chart is equivalent to '*false*'.

These errors in LSCs are not always as obvious as the one in Fig. 3(a) because such a cyclic dependency can involve many instance lines and large numbers of all kinds of LSC elements instead of just two instance lines like in Fig. 3(a). To support the actual authoring of quality LSC specifications, we relate the notion of well-formedness for LSCs from [8] to the concrete syntax and provide a fast algorithm that decides well-formedness on the syntactical representation of an LSC. Practically, LSCs should always be checked to be well-formed, in particular before expensive [5] checks like consistency of a whole LSC specification or LSC model-checking [11] are applied. This will significantly improve the usability of LSCs for formal verification because if the outcome of a model-checking run is not as expected, either because the LSC is insatisfiable or in the case of trivial satisfaction as outlined above, it is very helpful to know that the mismatch

between expectation and reality does *not* stem from the LSC not being well-formed so one can focus on the real reason for the mismatch.

The remainder of this article is structured as follows. Section 2 uses a small example to briefly recall the intuition of the LSC language of [11] and introduces abstract syntax and semantics using the new formalisation from [15]. Section 3 discusses the issue of non-well-formedness in more detail, formally defines well-formedness of LSCs, and exactly characterises the possible reasons for non well-formedness. Thereby we can justify that it is reasonable to consider only well-formed LSCs in this sense. In Sect. 4 we provide a fast algorithm that decides well-formedness on the abstract syntax, basically an acyclicity check on a particular structure, and discuss its complexity in terms of LSC size. Section 5 discusses the impact of well-formedness on related work, namely the complexity results of [5] that happen to be established just on the set of well-formed LSCs and it identifies LSCs played-in with the play-engine [9] to be well-formed by construction. Section 6 concludes and discusses further work.

2 Live Sequence Charts

To recall the syntax and intuition of LSCs, consider the LSC 'secure_crossing' given in Fig. 1(a). It states a high-level requirement on a distributed controller for a level crossing comprising a central controller 'CrossingCtrl' and separate controllers 'LightsCtrl' and 'BarrierCtrl' for traffic lights and barriers.

As mentioned in Sect. 1, the main difference between the LSC and the MSC language is that LSCs introduce modalities for whole charts, locations on instance lines, and LSC elements. The mode of the whole chart is indicated by the frame around the LSC body. The solid frame around the LSC body in Fig. 1(a) indicates that its mode is *universal*. A system satisfies a universal LSC if it adheres to the scenario *whenever* it is activated. The LSC in Fig. 1(a) is activated if the activation condition '*securing_request*' holds. In general, activation is characterised by a *pre-chart*, i.e. a prefix of the LSC body. The suffix of the LSC (the main-chart) is activated once the pre-chart has been completely observed. An activation condition is a shortcut for a pre-chart comprising only a single condition. A dashed frame around the LSC body indicates the other chart mode *existential*. A system satisfies an existential LSC if it has *at least one* run where the whole scenario (including the pre-chart) is observed *at least once*. This is the typical interpretation of MSCs.

The mode of a location in an LSC can be *hot* or *cold* and is indicated by the style of the instance line segment below the location. In Fig. 1(a) the topmost locations are cold on all instance lines but the one of the central controller 'CrossingCtrl', which is hot. The latter requires progress, that is, whenever the LSC is activated, a system only satisfies the LSC if the location is finally left by observing the element(s) at the following location. In the original proposal [8], the authors give the figurative intuition that one can't stand at a hot location forever without burning one's feet, thus one wants to leave it eventually. On the second location of instance line 'CrossingCtrl', the sendings of the instantaneous

messages '*lights_on*' and '*barrier_down*' are drawn at exactly the same location. They are thereby put in a *simultaneous region* (simregion for short) that requires them to both occur at the same point in time in a system that satisfies the LSC.

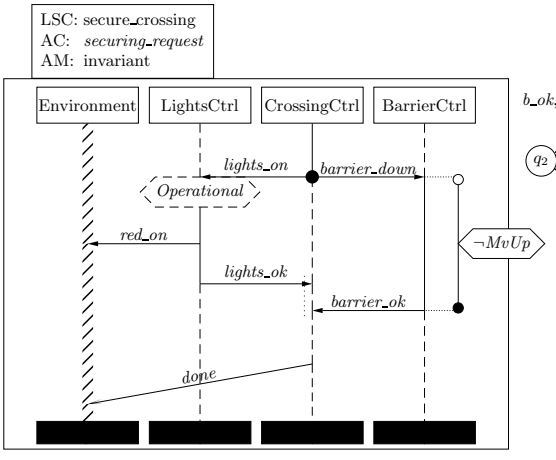
On the instance line of 'LightsCtrl', the reception of '*lights_on*' is co-located with a condition that has the mode *possible*, then also called cold condition. Thereby we express that if the lights are *not* operational, i.e. the cold condition doesn't hold at the point in time where '*lights_on*' is received, then the system needn't adhere to the rest of the chart. The chart is legally exited and immediately considered satisfied. Put the other way round, we do require that an implementation of the level crossing adheres to the remainder of the scenario whenever the lights controller is operational, i.e. if the cold condition holds. To specify requirements on the reaction of a non-operational lights controller we would provide another LSC that is activated in the situation where the lights controller is non-operational when receiving '*lights_on*'.

On the instance line 'BarrierCtrl', the reception of '*barrier_down*' is co-located with an exclusive beginning of a *mandatory* (or hot) *local invariant* that ends inclusively at the sending of message '*barrier_ok*'. By local invariants, requirements can be stated for spans of time in contrast to ordinary conditions which apply only to a single point in time. In the example we require that the barrier is not moving upwards immediately after '*barrier_down*' has been received up to and including the point in time where '*barrier_ok*' is sent. Should the barrier move upwards in this period of time, then the LSC is violated. Possible (or cold) local invariants are typically used to state assumptions. For example, if we change the lights controller's condition to a cold local invariant on the time between '*lights_on*' and '*lights_ok*' then we effectively say that a system has to adhere to the scenario unless the lights controller is not operational somewhere in between '*lights_on*' and '*lights_ok*'.

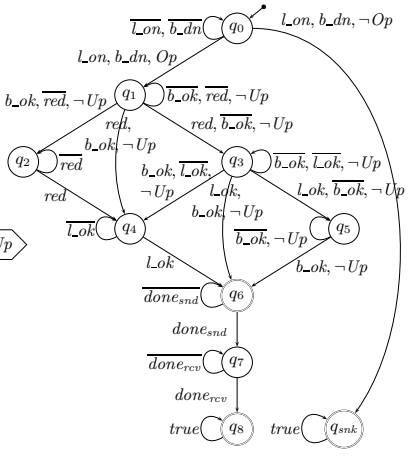
After receiving '*lights_on*', the lights controller is required to finally send a '*red_on*' message to the environment, that is, we decided that the traffic lights are not part of the model. If the lights are switched to red and the barriers are lowered, both distributed controllers should report success to the central controller by the messages '*lights_ok*' and '*barrier_ok*'. We indicate by the dotted line in parallel to the 'CrossingCtrl' instance line that we don't restrict the order of these two messages. They may occur in any order and even simultaneously. Such parts of instance lines where the order is explicitly relaxed are called *coregions*.

Note that the location before the sending of the asynchronous message '*done*', by which the central controller reports back that the crossing is secured, is cold and that we have then also reached cold locations on each other instance line. In this case, no progress is enforced. That is, a system that adheres to the LSC up to these locations but doesn't send '*done*' at all although satisfies the LSC.

In addition to giving the name and the activation condition, the *header* of the LSC in Fig. 1(a) comprises the *activation mode* that further restricts the activation. In order for a system to satisfy LSC 'secure_crossing' with activation mode *invariant*, each system run suffix that activates the LSC has to adhere to it. With activation mode *initial*, the LSC can be activated at most once per run,



(a) LSC for securing a level crossing.



(b) Symbolic Automaton $\mathcal{A}_L$ of secure_crossing's body (cf. Sect. 2.1).

$$\vec{I} \models_{LSC} L \Leftrightarrow \forall\, \vec{\iota} \in \vec{I}\, \forall\, k \in \mathbb{N}_0 : \vec{\iota}_k \models ac \Rightarrow \vec{\iota}/k \in \mathcal{L}(\mathcal{A}_L)$$

(c) Full Semantics of $L =$ 'secure_crossing' in terms of $\mathcal{A}_L$ (cf. Sect. 2.1).

Fig. 1. The securing protocol for a level crossing comprises switching on the red lights and lowering the barriers, each acknowledged by the responsible controller. For lack of space, message and condition names are abbreviated in Fig. 1(b), negation of message-observation predicates is expressed by overlining, and a comma is used for conjunction. E.g. q_0's loop fires if neither '*lights_on*' nor '*barrier_down*' are observed.

namely in the initial step. The third activation mode *iterative*, which excludes re-activation, lies outside the scope of this paper.

2.1 Syntax and Semantics

In the following, we introduce the syntax and semantics of LSCs following [11] in the formalisation of [15]. Note that we actually introduce a subset of LSCs that we call *core* LSCs. Core LSCs are missing three features that are out of the scope of this paper, namely we discuss activation only in form of activation conditions and not the general case of pre-charts, we exclude timer-set and -reset and timeout elements that LSCs inherit from MSCs, and for brevity don't consider possible messages, i.e. sending has to but reception needn't be observed. The first omission is not a restriction since the semantics of pre-charts is explained in terms of the same Symbolic Automaton construction used for main-charts thus our approach applies directly. The topic of timer consistency is orthogonal to the structural issues we discuss here.

One of the central informations of the concrete syntax of an LSC is the order of elements along a single instance line. As coregions may not be nested, the order of elements is actually a scenario order as defined in the following.

Definition 1 (LSC Instance Line). *Let P be a finite, non-empty set. The tuple $(P, \prec)$, $\prec \subseteq P \times P$, is called* instance line *iff $\prec$ is a* scenario order *(or* direct predecessor relation*) on P, that is, iff*

1. $\exists!\, a^\perp \in P\, \forall\, a \in P : a^\perp \prec^* a$ (Unique Minimum)
   where $\prec^*$ denotes the reflexive transitive closure of $\prec$.
2. $\forall\, a, a_1, a_2 \in P : a \prec a_1 \wedge a \prec a_2 \implies a_1 \not\preceq^* a_2$ (Unordered Successors)
   where $a_1 \not\preceq^* a_2$ denotes that a_1, a_2 are unordered*, i.e. $a_1 \not\prec^* a_2$ and $a_2 \not\prec^* a_1$.*
3. $\forall\, a_1, a_2 \in P : (\exists\, a_0 \in P : a_0 \prec a_1 \wedge a_0 \prec a_2)$
   $\implies (\forall\, a_3 \in P : a_1 \prec a_3 \implies a_2 \prec a_3)$. (Diamond Property)

A triple $(\mathbb{A}, \prec, \vartheta)$ with $\vartheta : \mathbb{A} \to \{hot, cold\}$ is called LSC instance line *iff $(\mathbb{A}, \prec)$ is an instance line. The elements $a \in \mathbb{A}$ are called* (tempered) atoms*. Two atoms $a_1, a_2 \in \mathbb{A}$ are called* instance co-located*, denoted by $a_1 \bowtie a_2$.* ♢

A core LSC is structured into the body and the information found in the head, namely the activation condition, the activation mode, and the quantification. The body is further structured into a set of LSC instance lines and three sets of the elements: messages, conditions, and local invariants. Each of these elements is equipped with an obligation mode from $\{mand, poss\} =: Obl$. Messages have a synchronity from $\{inst, asyn\} =: Sync$, and to each local invariant start- and end-atom a containedness from $\{incl, excl\} =: Cont$ is attached.

As the annotation of messages, conditions, and local invariants itself is not relevant in the course of this paper, we assume them to be from *Expr*, the set of boolean propositional expressions.

Definition 2 (Core LSC). *A* core LSC *is a tuple $L = (\ell, ac, am, quant)$ with* activation condition *$ac \in Expr$,* activation mode *$am \in \{initial, invariant\}$,* quantification *$quant \in \{existential, universal\}$, and $\ell = (Inst, Msg, Cond, LocInv)$ the* body *of the LSC where*

- *$Inst = \{(\mathbb{A}_1, \prec_1, \vartheta_1), \ldots, (\mathbb{A}_n, \prec_n, \vartheta_n)\}$ is a set of disjoint LSC instance lines. We set $Inst(L) := \{1, \ldots, n\}$, $\mathbb{A}_L := \bigcup_{i \in Inst} \mathbb{A}_i$, $\prec_L := \bigcup_{i \in Inst} \prec_i$, and $\vartheta_L := \bigcup_{i \in Inst} \vartheta_i$. We denote by $a_i^\perp$ the minimum of $\prec_i$, $i \in Inst(L)$, also called* instance head*, and set $\mathbb{A}_L^\perp := \{a_i^\perp \mid i \in Inst(L)\}$. By $A|_i := A \cap \mathbb{A}_i$ we denote the projection of a set $A \subseteq \mathbb{A}_L$ onto instance $i \in Inst(L)$.*

  If the LSC L is clear by context we shall simply write, e.g., $\prec$ instead of $\prec_L$.
- *$(m \in)$ $Msg =: Msg(L)$ is a set of* messages

  $$m = (a_s, a_r, \varsigma, \kappa, \psi_s, \psi_r) \in \mathbb{A} \times \mathbb{A} \times Sync \times Obl \times Expr \times Expr$$

  For each message $m \in Msg$ we set $atoms(m) := \{a_s(m), a_r(m)\}$.

  By $Msg_{inst}(L) := \{m \in Msg(L) \mid \varsigma(m) = inst\}$ we denote the set of instantaneous and by $Msg_{asyn}(L) := \{m \in Msg(L) \mid \varsigma(m) = asyn\}$ the set of asynchronous messages of L.

– ($c \in$) *Cond* =: *Cond*(L) *is a set of* conditions

$$c = (A_c, \kappa, \psi_c) \in 2^{\mathbb{A}} \setminus \{\emptyset\} \times \mathit{Obl} \times \mathit{Expr}$$

where there is at most one atom per instance line, i.e. $|(A_c|_i)| \leq 1$ *for* $i \in \mathit{Inst}$. *For each condition* $c \in \mathit{Cond}$ *we set* $\mathit{atoms}(c) := A_c(c)$;

– ($l \in$) *LocInv* =: *LocInv*(L) *is a set of* local invariants

$$l = ((a_s, \gamma_s), (a_e, \gamma_e), \kappa, \psi) \in (\mathbb{A} \times \mathit{Cont}) \times (\mathbb{A} \times \mathit{Cont}) \times \mathit{Obl} \times \mathit{Expr}.$$

For each local invariant $l \in \mathit{LocInv}$ *we set* $\mathit{atoms}(l) := \{a_s(l), a_e(l)\}$.

The set $\mathit{elems}(L) := \mathit{Msg}(L) \cup \mathit{Cond}(L) \cup \mathit{LocInv}(L)$ *is called the set of* elements of L. *The function 'atoms' is canonically extended to subsets of* $\mathit{elems}(L)$ *yielding sets of atoms. We set* $\mathit{atoms}(L) := \mathit{atoms}(\mathit{elems}(L))$. ♢

In order to formally capture well-formedness of LSCs in Sect. 3, we need to introduce a number of concepts belonging to the LSC semantics definition [11]. We stop with a brief introduction of the Symbolic Automaton that is the basis of the LSC semantics definition following [11] in order to be able to study the relation between the original definition of well-formedness and a semantical definition in terms of Symbolic Automata.

The central concept in the construction of the automaton is the *cut*, i.e. a set of atoms per instance line, indicating how far the LSC has been observed. A cut is empty or comprises at least one atom for each instance line. All instance co-located atoms in a cut are pairwise unordered.

Definition 3 (Cut). *Let* L *be a core LSC. A set of atoms* $\alpha \subseteq \mathit{atoms}(L)$ *is called* cut *iff*

1. $\alpha \neq \emptyset \implies \forall\, i \in \mathit{Inst}(L) : \alpha|_i \neq \emptyset$, *and*
2. $\forall\, i \in \mathit{Inst}(L)\ \forall\, a_1, a_2 \in \alpha|_i : a_1 \bowtie a_2 \implies a_1 \not\prec^* a_2$.

The empty cut is called initial cut of L *and denoted by* α_0, *the cut comprising all instance heads is called* instance heads cut of L *and denoted by* $\alpha_\perp(L)$, *and the maximal cut* α *with* $\forall\, a \in \alpha\ \forall\, a' \in \mathit{atoms}(L) : a \prec^* a' \implies a' = a$ *is called* final cut of L *and denoted by* $\alpha_{\mathit{fin}}(L)$.

The temperature of α *is 'cold' if* $\alpha = \alpha_{\mathit{fin}}(L)$ *or* $\vartheta(a) = \mathit{cold}$ *for all* $a \in \alpha$, *and 'hot' otherwise. The set of all cuts of* L *is denoted by* $\mathit{Cuts}(L)$. ♢

The unit by which a cut can be advanced is the simultaneous class (simclass for short). A simclass is defined by the fact that the two atoms of a synchronous message and all atoms of a condition are supposed to be observed simultaneously, and by transitivity, e.g. if two synchronous messages m_1, m_2 each use an atom which is also used by a condition c, then all atoms of these three elements m_1, m_2, and c belong to the same simclass.

Definition 4 (Simclass). *Let* L *be a core LSC. Two atoms* $a_1, a_2 \in \mathit{atoms}(L)$ *are called* simultaneous, *denoted by* $a_1 \sim a_2$, *iff*

1. $a_1 = a_2$, *or*
2. $\{a_1, a_2\} \subseteq \mathbb{A}_L^{\perp}$, *or*
3. $\exists\, e \in Cond(L) \cup Msg_{inst}(L) : \{a_1, a_2\} \subseteq atoms(e)$, *or*
4. $\exists\, a_3 \in atoms(L) : a_1 \sim a_3 \wedge a_3 \sim a_2$.

For each $a \in atoms(L)$ *we use* $[a]$ *to denote the equivalence class of 'a' wrt.* $\sim$*, i.e. the set* $\{a' \in atoms(L) \mid a' \sim a\}$*. The set* $atoms(L)/\sim$ *of all equivalence classes of atoms from* $atoms(L)$ *is also denoted by* $Simclass(L)$*, its elements are called* simclasses*. We use* $elems(scl)$ *to denote all LSC elements that share an atom with* $scl \in Simclass(L)$*, i.e. the set* $\{e \in elems(L) \mid atoms(e) \cap scl \neq \emptyset\}$. $\diamondsuit$

A simclass is intuitively enabled by a cut if each of its atoms either has its prerequisite in the cut or it belongs to a coregion and there is an atom in the cut from the same coregion. Furthermore the intuition of asynchronous messages is explicitly added in form of an additional restriction: reception of an asynchronous message shall be observed strictly after its sending.

Definition 5 (Enabled simclass). *Let* L *be a core LSC. Let* $\alpha \in Cuts(L)$ *be a cut of* L *and* $scl \in Simclass(L)$ *a simclass of* L*. A cut is said to* enable *'*scl*', denoted by* $\alpha \rhd scl$*, iff*

$$(\forall\, a' \in scl : prereq(a') \subseteq \alpha \vee \exists\, a \in \alpha : a \bowtie a' \wedge a \not\preccurlyeq^* a')$$
$$\wedge\ (\forall\, m \in Msg_{asyn}(L) \cap elems(scl) : a_r(m) \in scl \implies \exists\, a \in \alpha : a_s(m) \prec^* a)$$

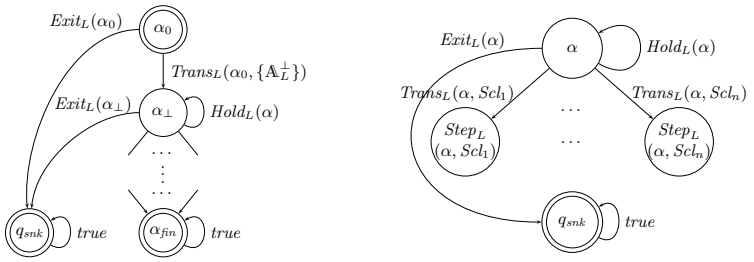
where $prereq(a) := \{a' \in atoms(L) \mid a' \prec a\}$ *is the* prerequisite *of* a*. The set of sets of simclasses* $Ready_L(\alpha) := \{\emptyset \neq Scl \subseteq Simclass(L) \mid \forall\, scl \in Scl : \alpha \rhd scl\}$ *is called the* readyset *of* α. $\diamondsuit$

Note that enabledness is a *structural* concept. It does not consider, e.g., the boolean expressions of conditions. In other words, the concept of enabledness can be seen as denoting *potential* progress. As there is no total order on the LSC elements, a single cut may enable multiple simclasses. Given a cut α and a non-empty set $Scl \in Ready_L(\alpha)$ of enabled simclasses from the readyset of α, the advancement function $Step_L(\alpha, Scl)$ denotes the (unique) follow-up cut.

We can now sketch the construction of an LSC's Symbolic Automaton which is basically a Büchi automaton where the transitions are labelled by boolean expressions. Formally, a *Symbolic Automaton* (SA) is a tuple $\mathcal{A} = (Q, q_s, \rightsquigarrow, F)$ comprising a finite set of states Q, the initial state $q_s \in Q$, the transition relation $\rightsquigarrow \subseteq Q \times Expr \times Q$, and the set of accepting states $F \subseteq Q$. We write $q_i \rightarrow q_j$ iff $(q_i, \psi, q_j) \in \rightsquigarrow$ for some ψ. An SA is called *Partially Ordered Symbolic Automaton* (POSA) if the reflexive transitive closure of $\rightarrow$ is a partial order.

The definition of an LSC's SA uses three kinds of predicates. The predicate $Hold_L$ on the self-loop characterises the condition under which a cut α is not advanced, $Exit_L$ on transitions to the legal exit characterises the conditions for legal exit from cut α, and $Trans_L$ corresponds to the observation of a given set of simclasses Scl in a particular cut α.

Definition 6 (Symbolic Automaton of an LSC). *The* Symbolic Automaton *of a core LSC* L*, denoted by* $\mathcal{A}_L$*, is the tuple* $(Q, q_s, \rightsquigarrow, F)$ *with* $Q = Cuts(L) \,\dot\cup\, \{q_{snk}\}$*,* $q_s = \alpha_0$*,* $F = \{\alpha \in Cuts(L) \mid \vartheta(\alpha) = cold\} \cup \{q_{snk}\}$*, and*



(a) Overall structure of $\mathcal{A}_L$. (b) Outgoing transitions from state α.

Fig. 2. Structure of the LSC body automaton. Double lined states are in F.

$$\begin{aligned}\rightsquigarrow = {} & \{(\alpha_0, \mathit{false}, \alpha_0)\} \cup \{(q_{snk}, \mathit{true}, q_{snk})\} \\ & \cup \{(\alpha, \mathit{Hold}_L(\alpha), \alpha) \mid \alpha \in \mathit{Cuts}(L) \setminus \{\alpha_0\}\} \\ & \cup \{(\alpha, \mathit{Exit}_L(\alpha), q_{snk}) \mid \alpha \in \mathit{Cuts}(L) \setminus \{\alpha_{fin}(L)\}\} \\ & \cup \{(\alpha, \mathit{Trans}_L(\alpha, \mathit{Scl}), \alpha') \mid \alpha \in \mathit{Cuts}(L), \mathit{Scl} \in \mathit{Ready}_L(\alpha), \alpha' = \mathit{Step}_L(\alpha, \mathit{Scl})\}\end{aligned}$$

Figure 2(a) shows the overall structure of the automaton, and Fig. 2(b) depicts the outgoing transitions of a single state resp. cut. Figure 1(b) gives the complete Symbolic Automaton of the LSC from Fig. 1(a). The Symbolic Automaton of an LSC is a POSA [15].

The notions introduced up to now are sufficient for the following sections that don't consider, e.g., information from the LSC header. For completeness, Fig. 1(c) exemplarily shows how the semantics of a complete LSC is expressed in terms of $\mathcal{L}(\mathcal{A}_L)$ the language accepted by the Symbolic Automaton of the body in [11]. The system model $\vec{I}$ is a set of *interpretation sequences*, that is, infinite sequences of interpretations of the predicates referred to by the LSC elements. It satisfies a universal invariant LSC L, denoted by $\vec{I} \models_{LSC} L$, iff each suffix $\vec{\iota}_k$ of a run $\vec{\iota}$ whose first snapshot $\vec{\iota}_k$ satisfies the activation condition is in the language of $\mathcal{A}_L$. In the initial activation mode, we would only consider $k = 0$.

3 Well-Formedness of Live Sequence Charts

Right from the original introduction of LSCs in [8] it has been clear that there are syntactically correct LSCs that shouldn't be considered legal. For this reason, [8] introduced a dependency relation on the LSC elements and restricted their definition of the semantics of LSCs to those with acyclic dependency relation. For the same reason, [5] restrict their investigation of the decidability and complexity of common problems, like consistency or realisability for a set of LSCs, to labelled partial orders thus they completely cover the well-formed LSCs of [8]. Up to now open is the question which syntactically correct *graphical* charts are actually ruled out by this restriction and how an author of LSCs best decides whether his LSCs are well-formed or not.

A first attempt to characterise well-formedness graphically to support authors of LSCs is given by [11]. They informally state, among others, the rule that

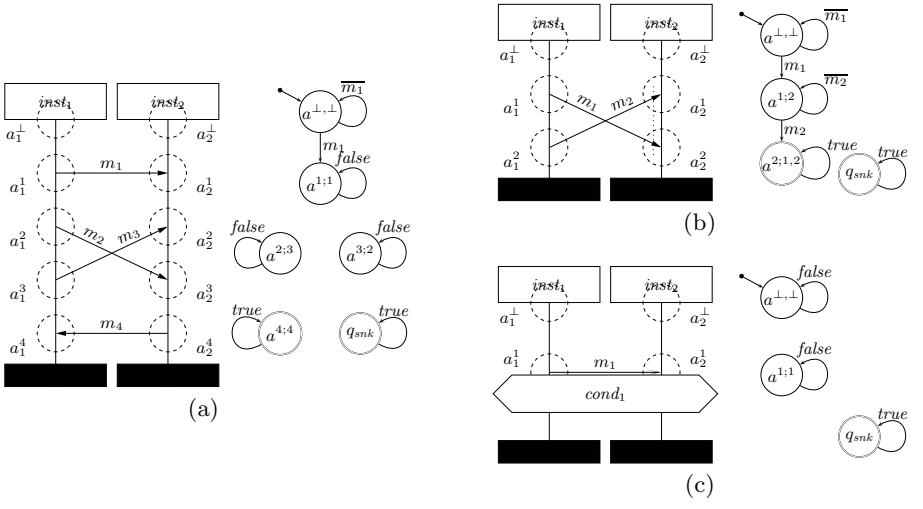


Fig. 3. Conflicting LSCs and their automata. For lack of space, we use $a^{x_1,\ldots,x\ ;y_1,\ldots,y}$ to denote the cut $\{a_1^{x_1},\ldots,a_1^{x}\ ,a_2^{y_1},\ldots,a_2^{y}\ \}$.

instantaneous messages shall not cross each other. This rule is intuitively safe. For example, the LSC shown in Fig. 3(a) is obviously not well-formed since messages m_2 and m_3 cyclically depend on each other. The rule correctly identifies it as such. But it incorrectly identifies the LSC shown in Fig. 3(b) as not well-formed although the cyclic dependency is broken using a coregion, i.e. this LSC actually is well-formed.

An example that is not covered by any rule of [11] is the LSC shown in Fig. 3(c). There the initial cut doesn't enable any subsequent simclass because the condition synchronises atoms a_1^1 and a_2^1 and message m_1 being asynchronous requires a_1^1, the sending, to be observed strictly before a_2^1, the reception.

In the following Lemma 1, we establish that there are exactly two possible reasons for an LSC not to be well-formed. Firstly, that the precedence imposed by the atom order contradicts the synchronisation imposed by instantaneous messages and conditions on their atoms. Secondly, that the precedence imposed by the atom order contradicts the order between the sending and reception atom of an asynchronous message. So by Lemma 1, the two cases shown in Fig. 3(a) and 3(c) are actually all. Another conclusion we can draw is that the language extensions of [11] don't introduce new means to produce non well-formed LSCs.

As these two cases are contradictions between fundamental principles of LSCs, they can't be resolved and thereby we can see that the restriction to well-formed LSCs is just right. We don't miss any interesting graphical representations of LSCs.

The following definition introduces the precedence relation $<_L$ that captures the interdependency between atoms on different instance lines based on the scenario order $\prec$, the asynchronous messages, and the synchronising elements like conditions and instantaneous messages.

Definition 7 (Precedence Relation). *Let L be a core LSC. For two atoms $a_1, a_2 \in atoms(L)$ we say that a_1* precedes *a_2, denoted by $a_1 <_L a_2$, iff*

1. $a_1 \prec^+ a_2$, *or*
2. $\exists\, m \in Msg_{asyn}(L) : a_1 = a_s(m) \wedge a_2 = a_r(m)$, *or*
3. $\exists\, a_1', a_2' \in atoms(L) : a_1 \in [a_1'] \wedge a_1' <_L a_2' \wedge a_2 \in [a_2']$. ♢

If the precedence relation is acyclic, then it is a strict partial order and its reflexive closure is the partial order $\leq_m$ that the discussion in [8] is restricted to. In terms of our precedence relation, well-formedness is defined as follows. Lemma 1 then introduces the two possible reasons for non well-formedness.

Definition 8 (Well-formed LSCs). *A core LSC is called well-formed iff its precedence relation $<_L$ is acyclic.* ♢

Lemma 1 (Contradiction). *A core LSC L is not well-formed iff*

1. $\exists\, scl \in Simclass(L)\ \exists\, a, a' \in scl : a \neq a' \wedge a <_L a'$ *(synchrony contradiction)*
2. *or* $\exists\, m \in Msg_{asyn}(L) : a_r(m) <_L a_s(m)$ *(asynchrony contradiction)* ♢

Proof. "$\Longrightarrow$": Let L be a non well-formed LSC. Then by Def. 8 the precedence relation has a cycle, i.e. $\exists\, a \in atoms(L) : a <_L a$ $(*)$.
As '$\prec^+$' is irreflexive, we are left with two reasons for $(*)$:

- There are atoms $a_1, a_1', a_2', a_2 \in atoms(L)$ s.t.

$$a <_L a_1 \wedge a_1 \in [a_1'] \wedge a_1' <_L a_2' \wedge a_2' \in [a_2] \wedge a_2 <_L a$$

  and $a_j \neq a_j'$, $j \in \{1, 2\}$. Then we have asynchrony contradiction for $[a_j]$.
- There are messages $m_1, \ldots, m_n \in Msg_{asyn}(L)$, $n > 0$, s.t.

$$a \prec^* a_s(m_1) \wedge a_r(m_1) \prec^* a_s(m_2) \wedge \cdots \wedge a_r(m_{n-1}) \prec^* a_s(m_n) \wedge a_r(m_n) \prec^* a$$

  For m_1 we have asynchrony contradiction.

Note that the cases are exhaustive but not exclusive. There are cycles of the precedence relation that show both contradictions.

"$\Longleftarrow$": For the other direction first consider case (2.). Then there is a message $m \in Msg_{asyn}(L)$ s.t. $a_r(m) <_L a_s(m)$. By Def. 7.2 we have $a_s(m) <_L a_r(m)$ thus a cycle. In case (1.) there is a simclass $scl \in Simclass(L)$ and atoms $a, a' \in scl$ with $a \neq a'$ and $a <_L a'$. Then there are atoms $a_1, \ldots, a_n \in atoms(L)$ s.t.

$$a \in [a_1] \wedge a_1 <_L a_2 \wedge a_2 \in [a_3] \cdots \wedge a_{n-2} \in [a_{n-1}] \wedge a_{n-1} <_L a_n \wedge a \in [a_n]$$

where $a_i <_L a_{i+1}$ by Def. 7.1 or 7.2 thus $a <_L a$ by Def. 7.3. □

Considering the Symbolic Automata of the LSCs in Fig. 3 shown next to the LSCs, we observe that the two non well-formed examples have in common that there are unconnected states in the automaton. Thus the particular Symbolic

Automata in Fig. 3(a) and 3(c) don't accept any run, they necessarily get stuck.

In the following, we establish that we could've equally well started to define well-formedness semantically by calling those LSCs well-formed whose Symbolic Automaton has a *traversable structure*. This is practically relevant for the tools which compile an LSC to its Symbolic Automaton that is then used for formal verification. Alternatively to applying the stand-alone well-formedness check introduced in Sect. 4 beforehand, these tools can speculatively start to construct the Symbolic Automaton and as soon as they hit a state with empty readyset testify that they are facing a non well-formed input. In addition, the construction of a cycle in the proof of Lemma 2 can then be used to determine an actual cycle and show it to the user.

Definition 9 (Traversable Structure). *Let $\mathcal{A} = (Q, q_s, \rightsquigarrow, F)$ be a Symbolic Automaton and $q \in Q$ state. The Symbolic Automaton $\mathcal{A}$ is said to have a* traversable structure wrt. *q iff q is reachable from any state that is reachable from the initial state, i.e. $\forall\, q' \in Q : (q_s \rightarrow^* q') \implies (q' \rightarrow^* q)$.* ♢

Lemma 2 (Well-formedness and Traversable Structure). *Let L be a core LSC and $\mathcal{A}_L = (Q, q_s, \rightsquigarrow, F)$ its Symbolic Automaton with $Q = Cuts(L) \dot\cup \{q_{snk}\}$. The LSC L is* well-formed *iff $\mathcal{A}_L$ has a completely traversable structure wrt. $\alpha_{fin}(L) \in Cuts(L)$.* ♢

For the proof of Lemma 2, we observe the following relation between synchrony and asynchrony contradiction and the enabling behaviour of legal cuts, i.e. cuts which are the result of successive application of the step function $Step_L$.

Lemma 3. *Let L be a core LSC.*

1. *Let $m \in Msg_{asyn}(L)$ be an asynchronous message with $a_r(m) <_L a_s(m)$. Then there is no legal cut $\alpha \in Cuts(L)$ with $\alpha \rhd [a_r(m)]$.*
2. *Let $scl \in Simclass(L)$ be a simclass s.t. there are $a, a' \in scl$ with $a \neq a'$ and $a <_L a'$. Then there is no legal cut $\alpha \in Cuts(L)$ with $\alpha \rhd scl$.* ♢

Proof. – Let $m \in Msg_{asyn}(L)$ with $a_r(m) <_L a_s(m)$. Assume there is a legal cut $\alpha \in Cuts(L)$ with $\alpha \rhd [a_r(m)]$. A reception is only enabled if the sending has been observed, i.e. there is $a \in \alpha$ with $a_s(m) \prec^* a$. Following backwards the precedence order chain $a_r(m) <_L a_1 <_L \ldots <_L a_n <_L a_s(m)$ there is an $a' \in \alpha$ with $a' \prec^* a_r(m)$, in contradiction to $a'' \not\succ^* a_r(m)$ or $a'' \prec a_r(m)$ for all $a'' \in \alpha$ as required by $\alpha \rhd [a_r(m)]$.

– Let $scl \in Simclass(L)$ s.t. there are $a, a' \in scl$ with $a \neq a'$ and $a <_L a'$. Assume there is a legal cut $\alpha \in Cuts(L)$ with $\alpha \rhd scl$. Following backwards the precedence order chain $a <_L a_1 <_L \ldots <_L a_n <_L a'$ yields $a \prec^* a_0$ for an atom $a_0 \in \alpha$ in contradiction to $\alpha \rhd scl$. □

Proof. (of Lemma 2)
"$\Longleftarrow$": (contraposition) Let L be a non well-formed core LSC and $\mathcal{A}_L$ its Symbolic Automaton. By Lemma 1, L has at least one contradiction:

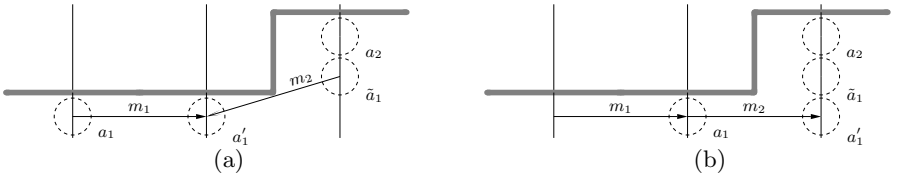


Fig. 4. Two reasons for $\alpha \not\rhd [a_1]$. The gray line shows the location of the cut α.

- *synchrony contradiction*, i.e. there is a a simclass $scl \in Simclass(L)$ and atoms $a, a' \in scl$ with $a \neq a'$ and $a <_L a'$. Then there is no cut which enables scl by Lemma 3.2.
- *asynchrony contradiction*, that is, there is an asynchronous message $m \in Msg_{asyn}(L)$ s.t. $a_s(m) <_L a_r(m)$. Then there is no cut which enables $scl := [a_r(m)]$ by Lemma 3.1.

If $\mathcal{A}_L$ had a completely traversable structure wrt. $\alpha_{fin}(L)$ then in particular $\alpha_0 \to^* \alpha_{fin}(L)$. Omitting the proof we use that in this sequence of cuts from α_0 to $\alpha_{fin}(L)$, for each atom $a \in atoms(L)$ there is a cut $\alpha \in Cuts(L)$ with $\alpha \rhd [a]$ in contradiction to the observation that scl is not enabled by any cut.

"$\Longrightarrow$": (contraposition) Let L be a core LSC whose Symbolic Automaton $\mathcal{A}_L$ is not completely traversable wrt. $\alpha_{fin}(L)$. As $\mathcal{A}_L$ is a POSA, there is a state $\alpha_{fin}(L) \neq \alpha \in Q = Cuts(L)$ that is reachable from the initial state $q_s = \alpha_0$ and doesn't enable any simclass, i.e. $Ready_L(\alpha) = \emptyset$. The cut α is not the initial cut α_0 because $\alpha_0 \rhd \alpha_\perp(L)$ by definition. By construction of $\mathcal{A}_L$ the cut α is reachable if it is the result of successive applications of the step function $Step_L$, hence α is legal. Let $A := \{a \in atoms(L) \mid \exists\, a_0 \in \alpha : a_0 \prec a \vee a_0 \not\asymp^* a\}$ be the set of atoms that are direct predecessors of an atom in α or belong to the same coregion as an atom in α. The set A comprises those atoms that can possibly be enabled by α. It is not empty because $\alpha \neq \alpha_{fin}(L)$

From A we can iteratively construct a cycle in the precedence relation as follows. Choose $a_1 \in A$. Its simclass $[a_1]$ is not enabled because α doesn't enable any simclass. There are two possible reasons for $\alpha \not\rhd [a_1]$ by the definition of '$\rhd$'. Firstly (cf. Fig. 4(b)) that there is an atom $a'_1 \in [a_1]$ that is a message reception and the sending $\tilde{a}_1$ has not yet been observed $(*)$ and secondly (cf. Fig. 4(a)) that there is an atom $a'_1 \in [a_1]$ whose prerequisite is not in α $(**)$, that is, $\exists \tilde{a}_1 \in atoms(L), a_0 \in A : a_0 \prec^+ \tilde{a}_1 \prec^+ a'_1$. As $\alpha \neq \alpha_0$, we can choose an atom $a_2 \in A$ with $\tilde{a}_1 \bowtie a_2$, i.e. instance co-located with $\tilde{a}_1$. If $\tilde{a}_1 \in A$, then the choice is $a_2 = \tilde{a}_1$. By $(*)$ or $(**)$ we have $a_1 <_L a_2$. For a_2 there is a similar choice of a'_2, $\tilde{a}_2$, and a_3 with $a_2 <_L a_3$. Etc.

Iterate this procedure until $a_n = a_1$, $n > 1$. The procedure terminates since A is finite and we directly have $a_1 <_L \ldots <_L a_n = a_1$. □

Note that well-formedness is not exactly equivalent to being insatisfiable by structure. In the case that the cyclic dependency occurs somewhere below a cold

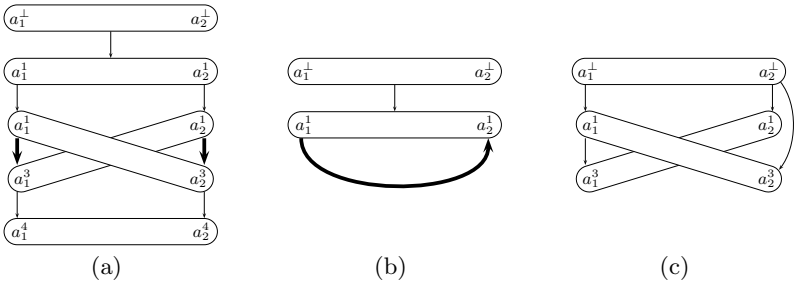


Fig. 5. Precedence Graphs of the three LSC from Fig. 3

cut of the main-chart, that is, not in the pre-chart, then a system that doesn't progress further than this cold cut properly satisfies the LSCs. Instead of refining Def. 9 to separate three cases, the well-formed and thus structurally satisfiable LSCs ("good"), the non well-formed and structurally insatisfiable LSCs ("bad"), and the non well-formed but satisfiable LSCs ("ugly") we note that for each "ugly" LSC there is a transformation to an equivalent "good" one.

4 Deciding Well-Formedness of LSCs

One result of the previous section was that well-formedness of LSCs can be checked on-the-fly when constructing its Symbolic Automaton. But all authors of LSCs need to know whether their specification is well-formed, not only the ones heading for formal verification where the Symbolic Automaton is constructed anyway. An independent mechanical check is evident from Definition 8: build up the precedence relation and check it for acyclicity. But the precedence relation is not the minimal data-structure for this purpose. Thus in the following we provide a faster well-formedness check based on the *precedence graph* of the LSC.

Definition 10 (Precedence Graph). *The* precedence graph *of a core LSC L is the directed graph $(Simclass(L), \hookrightarrow)$ with*

$$\hookrightarrow := \{(scl_1, scl_2) \in Simclass(L)^2 \mid \exists\, a_1 \in scl_1, a_2 \in scl_2 : \\ a_1 \prec a_2 \vee\ \exists\, m \in Msg_{asyn}(L) : a_s(m) \in scl_1 \wedge a_r(m) \in scl_2\}. \qquad \diamondsuit$$

The precedence graph structure reflects the order of simclasses along the instance lines and additionally tracks the dependency between the sending and reception of an asynchronous message. Fig. 5 shows the precedence graphs of the three LSCs from Fig. 3. Note that the two non well-formed LSCs exhibit cycles in Fig. 5(a) and 5(b), indicated by bold arrows.

Lemma 4. *A core LSC L is well-formed if its precedence graph is acyclic.* $\diamondsuit$

Proof. We show the equivalent claim that a cycle in the precedence relation of L implies a cycle in the precedence graph. Let $a \in atoms(L)$ with $a <_L a$, i.e. there are atoms $a_1, \ldots, a_n \in atoms(L)$ s.t.

$$a \in [a_1] \wedge a_1 <_L a_2 \wedge a_2 \in [a_3] \wedge \ldots \wedge a_{n-1} <_L a_n \wedge a \in [a_n]$$

where $a_i <_L a_{i+1}$ by Def. 7.1 or 7.2. Both cases entail $[a_i] \hookrightarrow^+ [a_{i+1}]$ by construction of the precedence graph. As $a_i \in [a_{i+1}]$ implies $[a_i] = [a_{i+1}]$ we can trace the complete precedence relation chain by the precedence graph relation and thus obtain $[a] \hookrightarrow^+ [a]$. □

Acyclicality of the graph of an LSC L is checked in $O(|Simclass(L)| + |\hookrightarrow|)$. The size of the graph relation $|\hookrightarrow|$ is bounded from above by $|\prec| + |Msg_{asyn}(L)|$. The number of edges induced by the scenario order is $\sum_{a \in \mathbb{A}} |prereq(a)|$. In the special case that L has no coregions, $prereq(a)$ is at most 1 thus $|\prec| \leq |\mathbb{A}_L|$. In the special case of only non-consecutive coregions we have $|\prec| \leq |\mathbb{A}_L| + |\mathbb{C}_L|$, where $\mathbb{C}_L \subseteq \mathbb{A}_L$ denotes the atoms of L that lie in a coregion. Only consecutive coregions produce a disproportional grow of $|\hookrightarrow|$ as then all possible combinations between the two coregions are reflected by $\prec$.

5 (Impact on) Related Work

Research into the LSC language is often restricted to the subset of well-formed LSCs, e.g. by abstractly representing an LSC in terms of a labelled partial order. The following subsection 5.1 discusses the impact of this restriction and extends existing results of well-formed LSCs to the set of all LSCs. Subsection 5.2 identifies all LSCs produced by the PlayEngine as being well-formed by construction.

5.1 LSCs as LPOs

As already mentioned in the previous sections, recent analysis of the complexity of Life Sequence Charts [3] uses as abstract syntax a labelled partial order, i.e. a tuple $(L, \leq, \lambda, A)$ where L is a finite set of events, $\leq \subseteq L \times L$ is a partial order on L, and $\lambda : L \to A$ is a labelling function, that is built as follows[1]:

Definition 11 (Order Relation of an LSC [3]). *Let L be a core LSC. Two simclasses $scl_1, scl_2 \in Simclass(L)$ are* directly ordered*, denoted $scl_1 <_d scl_2$, if*

1. $\exists\, a_1 \in scl_1, a_2 \in scl_2 : a_1 \prec a_2$, *or*
2. $\exists\, m \in Msg_{asyn}(L) : a_s(m) \in scl_1 \wedge a_r(m) \in scl_2$

The relation $\leq$ is the reflexive, transitive closure of $<_d$. ♢

The direct order between simclasses of Def. 11 corresponds to our precedence graph relation from Def. 10. Thus strictly speaking, the results of [3] only apply to the set of well-formed LSCs, not to the set of all LSCs since for non well-formed ones '$\leq$' is not a partial order by Sect. 3.

[1] Note that [3] consider neither simultaneous classes nor asynchronous messages. We extend their construction in the canonical way.

In the other direction we can conclude from Sect. 3 that for each well-formed LSC '$\leq$' is a partial order and thus [3] indeed applies to all practically relevant LSCs. To extend the complexity results to the set of all LSCs, including the non well-formed, we have to take the complexity of checking for well-formedness into account. Section 4 entails that the (in the best case polynomial) complexity classes identified by [3] are not left by checking for well-formedness beforehand.

5.2 Play-Engine LSCs

The PlayIn/PlayOut [9] approach employs Life Sequence Charts for specifying and executing behavioural requirements of reactive systems. LSCs are specified by "playing them in" on a prototypical GUI of the system, i.e. all interactions between the user and the GUI are recorded as LSCs. After a set of scenarios have been played in, the engine is also able to "play them out", i.e. when the GUI is operated the play-engine reacts according to the recorded specification.

Obviously, the user is not able to play in any contradictory LSC by using the GUI interface. Consequently, the results of Sect. 3 entail that every played-in LSC is well-formed by construction.

But in general it is not an option to exclusively use "played-in LSCs" in all application domains. For LSCs that are meant as requirement specification for an existing implementation, it is usually not appropriate to record every desired system run, but one rather wants to specify whole classes of scenarios at once. This is already supported by the play-engine in form of (very limited) editing capabilities. Also, the requirement that certain events happen simultaneously cannot be played in in a convenient manner.

6 Conclusion

The need to turn the abstract definition of well-formedness into something more imaginable to authors of LSC specification has already been identified by [11] and approached by an informal and incomplete list of guidelines. Our two alternative characterisations of well-formedness, in terms of concrete syntax and semantical in terms of the underlying Symbolic Automaton, provide for a better understanding of the relation between the set of syntactically correct LSC diagrams and the practically useful ones. Judging from the characterisations we can conclude that it is reasonable to restrict the discussion of LSCs to the ones that are well-formed in the sense of [8].

Further work comprises the sketched extension of the well-formedness notions and analyses to the whole LSC language of [11], in particular possible messages and timer-set/-reset and timeout elements. For LSCs with timing requirements it is also desirable to have fast syntactical sanity checks because the general case lies in the class of the timer-consistency problem of timed automata.

References

1. D. Amyot and A. Eberlein. An evaluation of scenario notations and construction approaches for telecommunication systems development. *Telecommunications Systems Journal*, 24(1):61–94, September 2003.
2. J. Bohn, W. Damm, H. Wittke, J. Klose, and A. Moik. Modelling and validating train system applications using Statemate and Live Sequence Charts. In *Proc. IDPT 2002*. Society for Design and Process Science, June 2002.
3. Y. Bontemps. *Relating Inter-Agent and Intra-Agent Specifications*. PhD thesis, University of Namur (Belgium), April 2005.
4. Y. Bontemps, P. Heymans, and H. Kugler. Applying LSCs to the specification of an air traffic control system. In *Proc. SCESM'03*, 2003.
5. Y. Bontemps and P.-Y. Schobbens. The complexity of Live Sequence Charts. In V. Sassone, editor, *Proc. FOSSACS 2005*, volume 3441 of *LNCS*, 2005.
6. A. Bunker, G. Gopalakrishnan, and K. Slink. Live Sequence Charts applied to hardware requirements specification and verification: A VCI bus interface model. *Software Tools for Technology Transfer*, 7(4):341–350, August 2004.
7. P. Combes, D. Harel, and H. Kugler. Modeling and verification of a telecommunication application using Live Sequence Charts and the play-engine tool. In *Proc. ATVA 2005*, number 3707 in LNCS, 2005.
8. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, July 2001.
9. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
10. ITU-T. *ITU-T Rec. Z.120: Message Sequence Chart (MSC)*. ITU-T, Geneva, 1999.
11. J. Klose. *Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior*. PhD thesis, C. v. O. Universität Oldenburg, 2003.
12. C. Knieke, M. Huhn, and U. Goltz. Modelling and simulation of an automotive system using lscs. In S. H. Houmb, J. Jürjens, and R. France, editors, *Proc. CSDUML'2005*. TUM, September 2005.
13. H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps. Temporal logic for scenario-based specifications. In N. Halbwachs and L. D. Zuck, editors, *Proceedings TACAS 2005*, volume 3440 of *LNCS*. Springer-Verlag, 2005.
14. I. Schinz, T. Toben, C. Mrugalla, and B. Westphal. The Rhapsody UML Verification Environment. In J. R. Cuellar and Z. Liu, editors, *Proc. SEFM 2004*, pages 174–183, September 2004.
15. T. Toben and B. Westphal. On the expressive power of Live Sequence Charts. In *Proceedings of the SofSem 2006 Poster Session*. Matfyz Press, 2006. To appear.
16. K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer. Scenarios in system development: Current practice. *IEEE Software*, 15(2):34–45, March 1998.

Concerned About Separation

Hafedh Mili[1], Houari Sahraoui[2], Hakim Lounis[1],
Hamid Mcheick[1], and Amel Elkharraz[1,2]

[1] LATECE, Université du Québec à Montréal, Montréal (QC), Canada
{hafedh.mili, hakim.lounis}@uqam.ca, hamid_mcheick@uqac.ca
[2] GEODES, Université de Montréal, Montréal (QC), Canada
{sahraouh, ekharraz}@iro.umontreal.ca

Abstract. The separation of concerns, as a conceptual tool, enables us to manage the complexity of the software systems that we develop. There have been a number of approaches aimed at modularizing software around the natural boundaries of the various concerns, including subject-oriented programming, composition filters, aspect-oriented programming, and our own view-oriented programming. The growing body of experiences in using these approaches has identified a number of fundamental issues such as what is a concern, what is an aspect, which concerns are inherently separable, and which aspects are composable. To address these issues, we need to focus on the semantics of separation of concerns, as opposed to the mechanics (and semantics) of aspect-oriented software development methods. We propose a conceptual framework based on a transformational view of software development. Our framework affords us a unified view of the different aspect-oriented development techniques which enables us a simple expression for the separability issue.

1 Introduction

"Separation of concerns" is a general problem-solving idiom that enables us to break the complexity of a problem into loosely-coupled, easier to solve, subproblems. Underlying this idiom is the hope that, 1) the subproblems are easier to solve, and 2) the solutions to these subproblems can be composed relatively easily to yield a solution to the original problem. The history of programming languages may be seen as a perennial quest for modularisation boundaries that best map (back) to "natural modularisation boundaries" of requirements. Aspect-oriented software development methods are no different. However, most of the research on AOSD has focused on the semantics of aspects and aspect composition, i.e. the solution domain, as opposed to the semantics of concerns and concern separation and composition, i.e. the problem domain. Yet, the early case studies have shown that these conceptually elegant techniques weren't intuitive to use (see [9], [8], [7]). Further, a great number of users of these techniques were caught up in the "how-to" of language constructs, with no regard for the conceptual appropriateness of the AOSD technique for the problem at hand. Further, the various techniques seem to offer orthogonal, but nonetheless useful constructs, with no clear guidelines (which method is appropriate for which problem).

We believe that better understanding of the AOSD techniques will result from a characterization of, 1) the *input* of software development, and 2) the *process* of

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 247 – 261, 2006.

software development, to help characterize, if not identify, which concerns are separable, and which development steps are most likely to affect the separation (or separability) of the resulting artifacts. We propose a conceptual framework based on a transformational view of software development. In this context, all the requirements on a software product, be they functional (related to input/output relations) or otherwise (related to *how* the output is produced), are inputs in these transformations. These requirements fit into general areas, or *concerns*, which may end up embodied in separate or same artifacts. We distinguish *essential separability* and *inseparability*, which characterize *requirements*, from *accidental separability* and *inseparability*, which characterize the *realizations* of those requirements in development artifacts. Accidental inseparability can be remedied by better language design and user education. Accidental separability should even be discouraged as the conceptual complexity is often increased, and maintenance of the resulting program is often made harder.

2 Understanding the Separation of Concerns Problem

Design is a very complicated cognitive task bringing to bear a host of knowledge types and sources and a myriad of problem solving skills [4]. When the artifacts, themselves, are complex, a number of the conceptual and methodological tools fall apart because of scalability problems. Many researchers have shown that complexity is an *essential* property of design activities *in general*, due in part to the *inevitably incomplete formulation* of the problem, and in part to our inability to cope *simultaneously* with all of the constraints of a given problem (our *bounded rationality* [16]).

The *separation of concerns* technique is a general problem solving heuristic that consists of solving a problem by addressing its constraints, first separately, and then combining the partial solutions with the expectation that, 1) they be composable, and 2) the resulting solution is nearly optimal. For this heuristic to yield satisfactory results, the concerns that we are trying to treat separately must be fairly independent, to start with, so that they don't interfere with each other. Further, the problem solving activity itself needs to yield solutions that are composable. In this section, we try to define the separation of concerns problem for the case of software. In this case, the "problem" is a set of requirements, and the "problem solving" process is the software development process. We first start by characterizing the software development process. In section 2.2, we try frame the separation of concerns problem.

2.1 A Transformational View of Software Development

Software development is a complex activity involving a variety of skills and a variety of conceptual and formal tools. For the purposes of reasoning about software development—and perhaps automating some of its steps— researchers and practitioners alike have found it useful to view software development as the process of going from specifications of what is to be done (requirements), to precise specifications of how it is to be done. Dasgupta identified two kinds of requirements in any design problem, *empirical requirements*, which specify externally observable or empirically determinable qualities that are desired of the artifacts, and *conceptual requirements*, which specify adherence to a particular style [4]. For the case of software, there are two

kinds of externally observable qualities, *functionality*—the *what*—on one hand, and run-time behavior—the *how*, including performance, and the like. Accordingly, we see three major categories of requirements for software development:

1. *Requirements of functionality*. These requirements specify an input/output relationship. To satisfy these requirements, we need a function that takes an input/output relationship and returns a function that returns the output for a given input
2. *Run-time requirements*. These are requirements on run-time behavior such as performance, distribution, the underlying machine (virtual or otherwise), etc.
3. *Requirements on the software artifacts*. These requirements deal with things such as modularity, reusability, choice of language, etc.

These correspond closely to the categories of architectural qualities identified by [2]. Describing a program using an executable specification language may be seen as performing a first step of the design process, i.e. ensuring functionality. Later steps can worry about run-time behavior and artifact quality. In practice, these three sets of requirements are addressed simultaneously. Further, except in new projects where a complete system is built from the ground up, new functionality often has to integrate into an existing architecture, which embodies a specific point in the design space that addresses a set of run-time and artifact requirements. However, for the purposes of our presentation, we will assume that the three major design dimensions are commutative; two design transformations T_1 and T_2 are said to be commutative if given D_i, the description of the software at step i, we have $T_2 \circ T_1 (D_i) = T_1 \circ T_2 (D_i)$ (see e.g. [3]). With this mind, let us propose a first-cut description of software development.

Handling functional requirements. Given a relation R: A×B, we need to obtain a function f: A → B, such that for all a ∈ A, f(a) ∈ ImageR(a). We say that f(.) is an implementation of R. R describes the relationship that must exist between the input and the output; f(.) provides an effective procedure for computing the output, given the input. If R(.,.) is not a function (i.e. some elements of A have more than one image), then f(.) picks one element. Automatic programming consists, to a great extent, of automating the "operationalization of requirements". This transformation may be described by a relation OR: {R}×{f(.)} from the set of relations to the set of functions. Let R be the set of relations and F the set of functions. OR is thus a subset of R×F. This relation may be known intensionally, or extensionally (through exemplar pairs). Automating this step consists of finding a function g: R → F such that given a relation R∈R, g(R(.,.)) = f(.) where (R , f(.)) ∈ OR. We say that g is an implementation of OR.

Handling run-time requirements. These include *performance requirements* and *execution model*. These requirements are handled differently from functional ones. Whereas the operationalization of requirements associates a requirement with *any* function that implements the requirement, here we are picky about the properties of such functions. For example, such functions have to be efficiently computed. Instead of the relation *OR* shown below, we now have a subrelation *EOR* (Efficient Operationalization of Requirements) such that *EOR* ⊆OR, where Domain (*EOR*) = Domain (*OR*), but $\text{Image}_{\text{EOR}}(R) \subseteq \text{Image}_{\text{OR}}(R)$. In other words, out of all the functions that implement R, we pick the ones that are efficient.

Issues related to the execution model include things such as distribution, synchronization, and security. This does not change the function that is computed but changes things about where the different pieces are executed and how. We can represent the execution of function f() as follows: EX: $\underline{F} \times I \times M \rightarrow O \times M$. EX takes three arguments: a) the function to be computed, b) its input(s), and c) the initial state of the machine. EX produces a pair of outputs: the result of applying the function to its input, and new state of the machine. In other words, EX (f(.), *i*, *s*) = <*f*(*i*), *s'*> where *s'* is the state of the machine after it has finished execution of function *f*(.) on input *i*. The state changes consist of the side effects of the execution and may involve things such as establishing or terminating connections, modifying the state of data on permanent storage, logging, collecting statistics, etc..

Generally speaking, EX is a composition of several functions. For example, with a virtual machine architecture, we have the hardware machine executing the virtual machine, and the virtual machine executing the actual program. The execution of the virtual machine itself could be written as VM (f(.), *i*, *s*) = <*f*(*i*),*s'*>. The hardware machine, in turn, is executing the function *VM* on its inputs, and changes state. The input of the *VM* consists of the triple <*f*(.), *i*, s>, where *f*(.) is the function to be executed—written in "virtual machine language"—*i* is the input of *f*(.), and *s* the state of VM. Let the hardware machine be represented by the function HM, we have HM (VM(.,.,.), (*f*(.), *i*, *s*), *hs*) = <<*f*(*i*), *s'*>, *hs'*>. And so forth. The virtual machine itself consists of a set of layered (composed) services or parallel services. An example of layered services is VM(f(.), *i*) = VM_1o VM_2(f(.),*i,s*). An example of parallel services is represented as < VM_1;VM_2>(f(.),*i*,*s*) where VM_1 and VM_2 are two services that are performed in parallel but such that the end result is the pair <*f*(*i*),*s'*>. It may be that one service computes the result (VM_1(f(.),*i*,*s*) = *f*(*i*)) while the other changes the state of the machine (VM_2(f(.),*i*,*s*) = *s'*). We could also have situations where VM_1 and VM_2 modify different parts of the state of the executing machine. The output itself may be computed by one or two of the virtual machines.

Handling requirements on the artifacts. This involves taking into account the packaging of the function *f*(.) based on a number of criteria, including a reasonable division of labor, reusability, cohesion and coupling of the resulting modules, etc. It also includes things such as the choice of a programming language, programming style, etc. Note that requirements on artifacts may lead us to implement *more* than the initial requirements. For example, reusability considerations may compel us to implement more generic classes to accommodate the needs of *other* applications within the same domain. It may also compel us to break down functions differently to identify common parts, without necessarily implementing more functionality than required. Let us take a problem R(.), and its realization, some function *f*(.). Idem for a problem R'() with realization *f'*(). If we can write $f = f_{post}$ o g o f_{pre} and $f' = f'_{post}$ o g o f'_{pre}, then we reduce the amount of new code to be developed.

2.2 Framing the Separation of Concerns Problem

For the purposes of our discussion, we define a *concern* as a set of related requirements. Elements of the set may be defined extensively (enumerated) or *intensively*, by referring to a domain (e.g. *security*). Simply put, requirements are sets of properties

that must be satisfied by the solution. If we use predicate logic to express requirements, a number of intuitions that we have about requirements have a simple expression in logic [21]. The basic premise of separation of concerns approaches to software development is that *requirements have nice properties, and to the extent that we can associate artifacts with concerns, we would like the artifacts to have similar properties*! Precisely, the "separation of concerns" methods rely on the existence of a development homomorphism such as the one illustrated in Figure 1. Assume that requirements are represented by predicates, and let $A_P = OR(P(.))$ be the artifact that corresponds to predicate $P(.)$. Development (represented by the thick arrow) is a homomorphism if there exists an operator $\oplus$ defined on artifacts such that $OR(P(.) \wedge Q(.)) \equiv OR(P(.)) \oplus OR(Q(.))$.

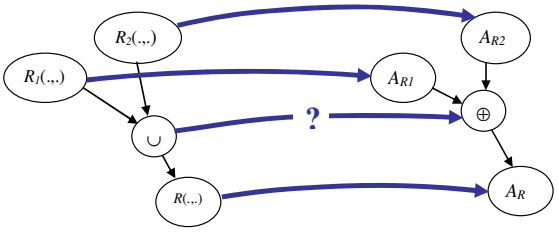


Fig. 1. Development is a homomorphism from requirements to artifacts

We have some intuitions about cases where this homomorphism between requirements and artifacts holds. For example, given two requirements defined by relations R_1: $A \rightarrow B$, and R_2: $B \rightarrow C$, we know of several operators $\oplus$ such that $OR(R_2 \circ R_1) \equiv OR(R_1) \oplus OR(R_2)$. For example, if the implementation adopts the call-and-return style, the operator $\oplus$ consists of the call relationship between procedures. If the publish-and-subscribe style is used, the operator $\oplus$ consists of registering $OR(R_1)$ as a publisher of some message, and $OR(R_2)$ as a subscriber to that message. Etc.

The advantages of this homomorphism include reusability, configurability, and separate maintenance. A number of object-oriented programming constructs and design idioms may be seen in this light. The new generation of separation of concerns techniques may be seen as defining new modularization boundaries for requirements, that are different from the ones afforded by regular object-oriented programming, and that are *realizable* in artifacts that are *composable* according to some composition operator. For example, OORAM uses *role models* [18] as new behavioral modules, and *role synthesis* to compose role models. Subject-oriented programming defined *subjects* [6] as new modular structures, and *subject composition*, as a composition mechanism [14]. Aspect oriented programming defines *aspects* as new module boundaries, and *aspect-weaving* as a way of composing aspects with regular classes [10]. Our own view-oriented programming uses *viewpoints* as a way of representing domain-independent business processes, and *view instantiation* and *attachment* as a way of adding that behavior to objects [12],[13]. All of these techniques may be collectively referred to as *aspect oriented development techniques,* where composition filters, subjects, aspects *à la* Kiczales et al., and our views may all be referred to as

aspects. Thus, we can talk about *functional aspects* which are associated with *functional concerns* or *architectural aspects* which are associated with *architectural concerns*.

Notwithstanding the case of OORAM, where the emphasis is on requirements level separation (*role models*) and composition, much of the so-called aspect-oriented development techniques have focused on the mechanics of artifact composition, sometimes losing sight of, 1) the requirements that these artifacts are supposed to embody, and 2) whether that composition (or separation) makes sense, from a requirements point of view. Further, even in those cases where AO techniques seemed appropriate, there were sometimes better non-aspect oriented solutions (see e.g. [15]).

If we view requirements as predicates on the solution, then requirements are clearly composable using logical composition ($\wedge$)—whether the resulting conjunction has solutions or not [21]. However, for the homomorphism of Figure 1 to hold, (1) the requirements that we need to compose have to be independent, and (2) the development transformations have to preserve such independence so that the resulting artifacts (aspects) may be combined.

3 All Concerns/Aspects Are Functional

We identified in section 2.1 three distinct kinds of requirements, requirements of functionality, run-time requirements, and requirements on the software artifacts themselves. Before we talk about the conditions under which different requirements (or concerns) may be separable, and whether we should try to untangle or compose their associated artifacts, we look for a common framework that would enable us to look at all three kinds of requirements, and that would enable us to take a simpler view of the separability and composability issue. We start our discussion by first characterizing the ways in which requirements in each category are handled (individually). We will argue that run-time requirements can be represented as functional requirements on the virtual machine; requirements on artifacts are more difficult to formalize.

3.1 Handling Run-Time Requirements

We consider run-time requirements to be functional requirements on an imaginary virtual machine that will execute the program in the context of the real machine. The virtual machine will add a number of services including distribution, persistence, security, and others. Persistence services may be seen as providing the program with an execution environment (a virtual machine) that persists automatically the objects that the program manipulates. Most object-oriented databases operate this way (Versant, ObjectStore): developers write programs that manipulate persistent objects in a seamless fashion. It is as if databases come with their own run-time object model, built on top of the host language object model. We later see how this is actually implemented—interestingly, a limited form of aspect-oriented programming.

Distribution is similar to persistence in principle. Lest we oversimplify, distribution may be seen as providing a virtual machine whose run-time representation of objects accommodates remote objects, with what that implies in terms of referencing and in

terms of method invocation. Consider the following CORBA or RMI-like code sequence:

```
Bank bank =
naming.bind("//www.mycompagny.com/mybusinessdomain/bank
23");
Client cl = bank.getCustomer("JohnDoe234");
String address = cl.getAddress();
```

Notwithstanding the first line, which suggests the use of a naming service, the subsequent lines are indifferent from the location of the objects. We could imagine the same program being run in local mode, where the default Java virtual machine runtime representation of objects is used, and "a distributed Java virtual machine" that uses a level of indirection for run-time object representation to access remote objects, and that invokes an ORB to execute methods. Existing implementations of distribution use a slightly different implementation but the idea is the same. In fact, some researchers have even attempted to *distribute* regular OO applications using AspectJ™ [17].

The way distribution and persistence have been commonly implemented present some commonality. Transparency to the developer dictates a virtual machine metaphor. However, both techniques instrument user code with service-specific code that invokes those services (persistence or remote access). With Java-style persistence (e.g. ObjectStore), the code that is injected is added directly to the compiled Java bytecodes. With distribution, the IDL compiler injects, along with user code, code that is meant to be executed by the distribution virtual machine.

The same can be said about some aspects of security. Both authentication and encryption can be easily (and naturally) implemented at the virtual machine level: one involves encrypting exchanged data (through method calls), and the other authenticates the caller. In fact, Java's own security model is supported and enforced by the virtual machine, which can be thought of as submitting method execution requests to a security manager. J2EE's security model is enforced by the containers—a higher level yet virtual machine.

One reason why virtual machine-like implementations of these services are not common—with the exception of security, for which we want no loopholes—is performance. The other is selectivity: because these services involve an overhead, if we embed it in the virtual machine, then all objects will use it, whether they need it or not. With this code injection mechanism, the code will only be injected in those objects/classes that need it.

As mentioned above, common implementations of persistence use a variant of aspect oriented programming: persistence code is added into designated class files (typically specified in configuration files) so that object creation, accessing, and modification access the database client. The same is true for distribution, where client-side stubs (proxies) go through the ORB to get the data they need. Viewing run-time requirements as functional requirements on the virtual machine helps us understand which services are separable and/or composable, and also helps us understand which solutions are feasible under which situations, and understand some of the anomalies that arise from composing virtual machine-level services.

3.2 Handling Requirements on the Artifacts

Requirements on the artifacts deal with development-time "abilities", with no regard for functionality or performance. Such requirements include understandability, reusability, maintainability, etc. Let $R(.)$ be a functional requirement, and $f(.)$ be an operationalization of $R(.)$, i.e. $f(.) \in OR(R(.))$. The various "abilities" on the artifacts can typically be written as constraints on various metrics on the artifacts, such as:

- $M_i(f(.)) = \text{MIN}_{g \in OR(R(.))} (M_i(g(.))$ (relative constraint) or
- $M_i(f(.)) \leq \alpha$, for some constant α (absolute constraint)

These *meta-level* constraints determine the packaging of the functionality.

Separation of concerns *is* a requirement on software artifacts that is being addressed with AOSD techniques. Thus, our discussion of how development affects separation of concerns will be limited to the development activities related to accommodating functional requirements and those related to handling run-time requirements.

3.3 Concerns Are Functional While Aspects May Not Be

Notwithstanding requirements on the artifacts themselves, we have functional requirements and run-time requirements. Run-time requirements are either measurable quality constraints (e.g. performance, space usage), or architectural services. We have shown in section 3.1 that the latter may be thought of as functional requirements on the virtual machine that executes the program. If we take this view, we could view both SOP and AOP, say, as being *both* concerned with the composition of *functional* concerns (or the corresponding aspects), with the difference that:

1. SOP (and our own method, VOP) manipulates functional concerns and aspects of the user program directly
2. AOP translates functional concerns on the virtual machine that executes a program P, into non-functional aspects to be woven into program P.

In fact, a number of researchers have recognized that the kind of concerns that AOP handles well are best (most simply) expressed at the meta-level, and a number of successors to Kiczlaes's AOP use a meta-level architecture to *add functionality* to the way these machines execute programs—mostly for intercepting message sends to perform processing before or after. In fact, Kiczales himself has said on many occasions that he developed AspectJ™ as a more constrained/safer version of the MOP to enable "average developers" to add pervasive behavior without compromising the integrity of the VM. Filman & Friedman consider *quantification* and *obliviousness* as essential features of AOP [5]. Both properties can be naturally expressed at the virtual machine level. Steimann that there are no aspects *à la* AspectJ™ for domain models [19]: aspects are solution (read: software) artifacts, and should have no place in domain models or in object-oriented analysis.

If we accept that aspects *à la* AspectJ™ are functional aspects on the virtual machine, we can immediately see that functional concerns and run-time concerns are orthogonal, and we can address them separately, at least up to the analysis step. We can also see that we shouldn't even try to combine functional aspects of the program(s) that we are developing with functional aspects of the machines that execute

them! At least not conceptually. And yet, that is what AspectJ™'s weavers were explicitly created for!

Composition filters are based on the message passing (and interception) metaphor, but the filters can be either functional, in which case we deal with the normal functional composition, or architectural, in which case, they too, could be handled at the virtual machine level. Thus, for the purposes of understanding the separability/composability of requirements, and the corresponding composability of the associated software artifacts, we need only to focus on the functional separately or composability of (functional) requirements.

4 Characterizing the Separability of Requirements

In this section, we attempt the overly ambitious goal of answering two dual questions:

1. Given two requirements, under what conditions can they be "developed" separately, and can their realizations (aspects) be composed at will. The answer to this question will help determine the *domain* or *operating range* of the development homomorphism we illustrated in Figure 1. We refer to this problem as the *composability of requirement realizations*.
2. Given a realization that addresses several concerns, under what conditions can that realization be untangled into separable aspects, each of which addressing a subset of concerns. The answer to this question may help us assess which systems may be re-engineered in such a way that different concerns are addressed in separate—and readily reusable—aspects. We refer to this problem as the *separability of requirement realizations*.

In addition to its practical importance, an answer to the second question will also help us understand why case studies have not been as convincing as the textbook cases that the original method authors have presented in support of their techniques.

Section 4.1 looks at the *composability of requirement realizations* problem for the case of functional requirements. We examine the problem from a purely mathematical point of view, reducing the separability of two requirements, seen as (input,output) relations, to conditions on their domains and ranges. This will enable us to address composability issues between runtime requirements or between functional requirements, but not between a functional requirement and a run-time requirement. Section 4.2 tries to answer the *separability of requirement realizations* for functional requirements by looking at the problem of decomposing a function into separate subfunctions. We look at a range of decomposition/recomposition operators with different semantics preserving properties.

4.1 Composable Requirements

Given a development transformation T, we consider two requirements R_1 and R_2 to be T-composable if:

1. we can associate separate realizations to them ($T(R_1)$ and $T(R_2)$), and
2. there exists a composition operator $\otimes$ on their realizations that satisfies them both, i.e. $T(R_1 \wedge R_2) = T(R_1) \otimes T(R_2)$

We showed in section 2.1 that functional requirements are transformed using an *operationalization operator—OR*, turning an input-output relation into a *function* that produces the output given the input. Having argued in section 3.1.1 that run-time requirements are nothing but functional requirements on the virtual machine, we look at the problem of composing two functional requirements through the operationalization operator.

We would like the operationalization of functional requirements to be additive at least in those cases where the two requirements have disjoint domains. Consider two relations R and R' such that Domain(R) $\cap$ Domain(R') $= \Phi$.The simplest way of implementing R$\cup$ R' is by taking $f(.)\oplus f'(.)$, where $f(.)\oplus f'(.) = g(x)$ such that:

$$g(x) = f(x), \text{ if } x \in \text{Domain}(R)$$
$$= f'(x), \text{ if } x \in \text{Domain}(R')$$

In other words, the simplest *OR*(.) would behave as $OR(R\cup R') = f(.)\oplus f'(.)$

Note that if we take into account reuse, then we may be able to write $f = f_{post}$ o g o f_{pre} and $f' = f'_{post}$ o g o f'_{pre}. We do have Domain(f'_{pre}) = Domain (f') and Domain(f_{pre}) = Domain (f), and thus Domain(f_{pre}) $\cap$ Domain(f'_{pre}) = Φ, but we don't know whether Domain(f_{post}) and Domain(f'_{post}) are disjoint, and we can't write *OR*(R$\cup$ R') (or $f(.)\oplus f'(.)$) as $[f_{post}(.)\oplus f'_{post}(.)]$o g o $[f_{pre}(.)\oplus f'_{pre}(.)]$.

If the relations have intersecting domains, we can define them as follows: $R = R_1\cup R_2$ and $R' = R'_1\cup R'_2$ such that: Domain(R_1) = Domain(R) - Domain(R'), Domain(R'_1) = Domain(R') - Domain(R), and Domain(R_2) = Domain(R'_2) = Domain(R) $\cap$ Domain(R'). In this case, the relation to implement is $R_1 \cup R'_1\cup (R_2\cup R'_2)$, where R_1, R'_1, and $R_2 \cup R'_2$ have mutually disjoint domains. Thus, we have $OR(R_1 \cup R'_1\cup (R_2\cup R'_2)) = OR(R_1) \oplus OR(R_2) \oplus OR(R_2\cup R'_2)$.

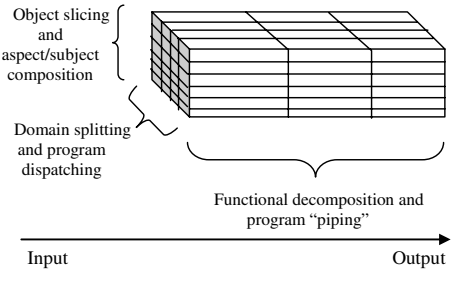


Fig. 2. Comparing three decomposition paradigms

This relationship is trivially satisfied in case $R_2 = R'_2$. This is the ideal case in the sense that both requirements agree on what the output should be for the same inputs. In that case, the two requirements (R_1 and R_2) may be seen as two restrictions of the same relationship defined on the domain Domain(R_1) $\cup$ Domain(R'_1). If the two relationships disagree on the output, then we have a problem. We see two levels of disagreement. The first level of disagreement is illustrated in the following example. Consider the two relations $R_1 = \{ (x,y) \mid 0 < x < 100, \text{ and } x^2 = y\}$ and $R_2 = \{ (x,y) \mid 50 <$

$x < 150$, and $x^2 = y$}. The intersection of the two domains consists of the interval [50..100]. If both the realizations of *R1* and *R2* use the positive square root of x—or both use the negative square root—then we are fine. If they use different square roots, then we have a problem. This incompatibility is due to an inconsistent choice of realizations, and is a common and acceptable course of action. Intuitively, what we need in this case to make sure that we use consistent realizations. This is not unlike the problem of choosing consistent specializations when we instantiate a framework, i.e. the kind of situations for which things such as the *factory pattern* is applicable.

The second level of disagreement is the case where the requirements themselves disagree, i.e. $\exists\, x \in \text{Domain}(R_1) \cap \text{Domain}(R_2)$ s.t. $R_1(x) \neq R_2(x)$

In our view, this is not a case for separation of concerns methods to handle: the requirements disagree, so there is no point in trying to compose the artifacts.

4.2 Separable Requirements

Given a development transformation T, we consider a requirement R (an element of the domain of T) to be *T-separable* if there exist, 1) two requirements R_1 and R_2, 2) a composition operator $\bullet$ defined on the domain of T—the requirements—and, 3) a composition operator $\otimes$ on the image of T—the *artifacts*—such that:

1. $R = R_1 \bullet R_2$
2. $T(R) = T(R_1) \otimes T(R_2)$

This is the good old divide-and-conquer analytical development paradigm. With structured analysis and design (and programming), the operator is functional composition, in the mathematical sense, and $\otimes$ is "piping", in the programming sense (the output of a program or procedure is used as an input to the other). Functional decomposition is not only useful for reducing complexity, it is also useful for reuse.

Another valuable pair of operators corresponds to the combination of domain splitting and dispatching. Consider the requirement R where $\text{domain}(R) = D = D_1 \oplus D_2$.—the symbol $\oplus$ referring to disjoint union (partition). Let T be the operationalization of requirements ($OR(.)$), and $R_1 = R|D_1$, and $R_2 = R|D_2$. Then:

$$OR(R(.)) = \begin{cases} \textbf{if } x \in D_1 \textbf{ call } OR(R_1) \\ \textbf{if } x \in D_2 \textbf{ call } OR(R_2) \end{cases}$$

We are all familiar with these two techniques, and have used them—and should continue to do so—to good measure. Aspect-oriented development techniques advocate *other* pairs of decompose/recompose or split/join operators which are specific to the object-oriented context. These new pairs of operators operate simultaneously on functions and data, along the lines of object or class hierarchy slicing (see e.g. [20]). In this case, instead of considering the input domain (D) as consisting of simple value, we consider it as a tuple (of state variables), and functions (object methods) may operate on various "sub-tuples".

Figure 2 illustrates the three decomposition paradigms. For each paradigm, we mention the decomposition technique used on requirements, and the corresponding composition technique used on the corresponding artifacts. Now, we look more

closely at the problem of sliceability of requirements. We start with a strict definition of sliceability which supports unrestricted (commutative) recomposition of the artifacts. We then propose a weaker form of sliceability which requires an ordered (non-commutative) recomposition.

Sliceability. Let R $\subseteq$ A×B, let f(.) = OR(R), and assume that A = S1 ×S2 ×… ×Si ×Si+1 ×… ×Sn×I and B = S1 ×S2 ×… ×Si ×Si+1 ×… ×Sn×O. We say that R (or f()) is sliceable if there exist two functions f1(x1,…,xi,i) et f2(xi+1,…,xn,i) such that f(x1,…,xi,xi+1,…,xn,i) = f1(x1,…,xi,i) •f2(xi+1,…,xn,i). In other words, the function f() can be computed as the concatenation of two functions.

The idea of sliceability is related to the idea that a relation may be written as a subset of the product of two relations. For example, let R_1 and R_2 be two binary relations. We can define the relation $R_1 \times R_2$ as follows: $<x_1,x_2,y_1,y_2> \in R_1 \times R_2$ if and only if $<x_1,y_1> \in R_1$ and $<,x_2,y_2> \in R_2$.

Intuitively, the sliceability corresponds to the case where we have two functions that take the same input and that use and modify different parts of an object, i.e. they correspond to two disjoint slices of the same data (or object). Sliceable functions can be put together, with no problem. Notice that we require that both functions take the input (which may be either a real input or a method selector), and that the output is produced between them. In the context of an object-oriented program, if we have a method that returns void but modifies the state of the object, then each subfunction will have modified its slice. If the function returns a value, then we might be able to find a subset of state variables based on which the output is computed, and the slice may be made along that. Note, however, that not all relations/functions are sliceable. A function that averages the state variables will not be sliceable.

Subject-oriented programming (and hyperspaces) works best with this ideal case in mind. Problematic cases occur when the sliceability hypothesis fails. Interestingly, the broken delegation problem can be understood in terms of sliceability of functions. Broken delegation happens when a function that occurs on one side (i.e. in a single object fragment) calls a separable function that occurs on several object fragments (see e.g. [1]): the result is no longer separable.

Effective sliceability. Let R $\subseteq$ A×B, let $f(.) = OR(R)$, and assume that A = $S_1 \times \ldots \times S_i \times \ldots \times S_j \times \ldots \times S_n \times I$ and B = $S_1 \times \ldots \times S_i \times \ldots \times S_j \times \ldots \times S_n \times O$. Let $f(\ldots)$ be a function that implements R. Let $f_1(\ldots)$ and $f_2(\ldots)$ be two functions with domains $S_1 \times \ldots \times S_i \times \ldots \times S_j \times \ldots \times S_n \times I$. If $f(x_1,\ldots,x_i,x_{i+1},\ldots,x_n,i) = <x'_1,\ldots,x'_i,x'_{i+1},\ldots,x'_n,o>$, we use the notation $f_{|i}$ to refer to the projection of f over the set S_i, i.e., $f_{|i}(x_1,\ldots,x_i,x_{i+1},\ldots,x_n,i) = x'_i$. Similarly, we define $f_{|S}$ as the projection of f over the set $S = S_i \times \ldots \times S_j$ for some i and j. Let Ref(f) be the set of variables used in the computation of $f(\ldots)$ and Mod(f) be the set of variables modified by $f(\ldots)$ be the set of state variables that are modified by f, i.e. the set of variables $\{ x_i\}_i$ such that $f_{|i}(x_1,\ldots,x_i,x_{i+1},\ldots,x_n,i) = x'_i \neq x_i$. A function $f(\ldots)$ is said to be *effectively sliceable* if and only if there exist two functions $f_1(x_1,\ldots x_n,i)$ and $f_2(x_1,\ldots x_n,i)$ such that:

$$\text{Mod}(f_1) \cap \text{Ref}(f_2) = \Phi,\ \text{Mod}(f_2) \cap \text{Ref}(f_1) = \Phi,\ \text{Mod}(f_1) \cap \text{Mod}(f_2) = \Phi$$
$$\text{Mod}(f_1) \cup \text{Mod}(f_2) = \text{Mod}(f),$$

$$f_{|\,Mod(f)}\,(x_1,\ldots,x_n,i) = f_{1\,|\,Mod(f1)}\,(x_1,\ldots,x_n,i) \bullet f_{2\,|\,Mod(f2)}\,(x_1,\ldots,x_n,i), \text{ and}$$
$$f_{|\,o}\,(x_1,\ldots,x_n,i) = f_{1\,|\,o}\,(x_1,\ldots,x_n,i) \bullet f_{2\,|\,o}\,(x_1,\ldots,x_n,i)$$

for some ordering of the state variables $x_1,\ldots,x_n$. Figure 3 illustrates the first three equalities in a Venn Diagram. Note that a *sliceable* function is also *effectively sliceable*. An interesting property of *effectively sliceable* functions is that the component functions may be executed in any sequence. There are other cases of sliceability, but in this case, the subfunctions have to be executed in a particular order. We call this *temporal sliceability*. Temporal sliceability is a weaker condition than effective sliceability, and is described as follows. Let $R \subseteq A \times B$, let $f(.) = OR(R)$, and assume that $A = S_1 \times \ldots \times S_i \times \ldots \times S_j \times \ldots \times S_n \times I$ and $B = S_1 \times \ldots \times S_i \times \ldots \times S_j \times \ldots \times S_n \times O$. Let $f(\ldots)$ be a function that implements R. Let $f_1(\ldots)$ and $f_2(\ldots)$ be two functions with domains $S_1 \times \ldots \times S_i \times \ldots \times S_j \times \ldots \times S_n \times I$. Using the same notation as above, we say that function $f(\ldots)$ is said to be *temporally sliceable* if and only if there exist two functions $f_1(x_1,\ldots x_n,i)$ and $f_2(x_1,\ldots x_n,i)$ such that:

$f_{|\,Mod(f)}\,(x_1,\ldots,x_n,i) = f_{1\,|\,Mod(f1)-Mod(f2)}\,(x_1,\ldots,x_n,i) \bullet f_{2\,|\ A-(Mod(f1)-Mod(f2))}\,(f_1(x_1,\ldots,x_n,i))$.

Mod(f_1) - Mod(f_2) represents the set of variables that are modified by f_1 but not by f_2. Some of these variables may, however, be referenced by f2 and we don't care about that. Obviously, the relationship between f_1 and f_2 is not a symmetrical one, and the functions have to be executed in a particular order.

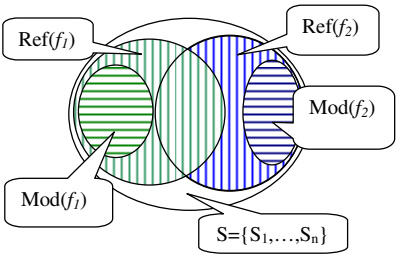


Fig. 3. A function is *effectively sliceable* if it can be written as the concatenation of two functions that modify disjoint parts of an object, and don't refer to the parts that the others modify

In [11], we showed that provided that methods of objects do not modify objects other than the executing ones, any method that computes a function and modifies the receiver object can be decomposed into a sequence of pure functional and purely side-effectal functions. To compose two hybrid functions, we decompose them along the purely functional versus purely side-effectal dimensions, find the smallest granularity decomposition between the two, and then compose them slice-by-slice.

The major problem, of course, is our tendency to code "service-oriented functions", i.e. functions that are application level but that are coded at the domain class level. These functions are not composable because they address an application specific need, each. You would want to compose them because they embody a general behavior that is not encapsulated elsewhere. Obviously, not choosing the right granularity is a problem, and leads to methods that are not composable.

5 Discussion

This is a very preliminary investigation into the principles of separation of concerns and the foundations of the techniques that promote separation of concerns. The yardstick by which innovations in software engineering are to be assessed has always been—and rightly so—to determine the problem that a given method, technique, or tool, solves. Separation of concerns is only useful to the extent that once the concerns have been addressed separately, we are able to re-combine the individual and partial solutions into one that addresses all of them.

Some of the case studies that are available in the literature show cases where concern separation is difficult in practice [7], [8], [9]. Others showed that *aspect/subject* composition is difficult, even in cases where the aspects or subjects embody distinctly different concerns [9], [11], [15]. We attempted to frame the separation of concerns in software development in terms of homomorphisms of development transformations, and then we tried to determine the "operating range" of these homomorphisms. This preliminary work raised more questions than it answered, and some of the answers are reassuringly common-sensical, but are worth stating:

- Not all requirements (concerns) are composable in the sense that they lead to composable artifacts. Viewing requirements as input-output relations, we identified simple conditions on the domains and images of these requirements, which essentially say that the requirements should not be conflicting. In particular, method cancellation through subject composition or aspect weaving is no less dangerous than cancellation with inheritance: they are both a sign of either a violation of intent, or of sloppy realization (implementation).
- We should treat aspects that embody run-time requirements differently—and separately—from aspects that embody functional (domain) requirements. We framed run-time requirements (persistence, fault-tolerance, etc.) in terms of *functional requirements of the virtual machine*. In an ideal world, such concerns should also be handled by virtual machine—or more generally, meta-level—aspects, and a number of recent approaches have gone that route. However, other considerations, such as performance, security, integrity, or portability, may suggest otherwise at the risk of inducing composability problems.
- Not all programs that implement several concerns can be reengineered into separate aspects. The underlying concerns/requirements may not be separable (essential inseparability), or the current implementation may not lend itself to such a separation (accidental inseparability). Object slicing helps with accidental inseparability.

We have started to take a closer look at the existing AOSD methods and the case studies to judge the usefulness of the above framework. We were able to explain known difficulties with subject-oriented composition (see e.g. [14]) and view attachment [13] in terms of violations of some of the principles outlined above. We have also started looking at the broken delegation problem from the perspective of functional composability and separability. The broken delegation problem manifests itself when we use aggregation (and message forwarding) as a way of compose behaviors. The problem is often referred to as an all-or-nothing problem. The "self" in an object component is either used to refer to the component—in which case we have broken delegation—or to the entire object, in which case, we do not have a problem. We have

shown elsewhere that attempts to fix the broken delegation problem can seriously compromise application security, and what we need is a more analytical approach to the problem. Our approach enables us to frame the problem.

References

1. D. Bardou & C. Dony, “Split objects: a disciplined use of delegation within objects,” *in proc. OOPSLA’96*, 1996.
2. L. Bass, P. Clements & R. Kazman, *Software Architecture in Practice*, 1998.
3. I. Baxter, “Design Maintenance Systems,” *CACM*, vol. 35, no. 4, April 1992, pp. 73-89.
4. S. Dasgupta, “The Nature of Design Problems,” in *Design Theory and Computer Science*, Cambridge University Press, 1991, pp. 13-35.
5. R. E. Filman & D. P. Friedman, “Aspect-oriented programming is quantification and obliviousness,” in *proc. of OOPSLA workshop on Advanced Separation of Concerns*, 2000.
6. W. Harrison & H.Ossher, “Subject-oriented programming: a critique of pure objects,” *in Proc. OOPSLA’93*, 1993.
7. S. Herrmann & M. Mezini, “On the Need for a Unified MDSOC Model: Experiences from Constructing a Modular Software Engineering Environment”, MSDOC workshop, OOPSLA’00, 2000
8. E. A. Kendall. *Role Model Designs and Implementations with Aspect Oriented Programming*. *In Proc. OOPSLA’99*, 1999
9. M. Kersten & G. Murphy, “Atlas: a case study in building a Web-based learning environment using aspect-oriented programming", in *proc. OOPSLA’99*, 1999.
10. G. Kiczales, J. Lamping, C. Lopez, “Aspect-Oriented Programming,” in *Proc. ECOOP’*97.
11. H. Mili, “On behavioral descriptions in oject-oriented modeling”, *Journal of Systems and Software*, 1996.
12. H. Mili, A. Mili, J. Dargham, O. Cherkaoui & R. Godin, "View Programming: Towards a Framework for Decentralized Development and Execution of OO Programs," *Proceedings of TOOLS USA ‘99*, 1999.
13. H. Mili, H. Mcheick & J. Dargham, ‘‘CorbaViews: Distribting objects with several functional aspects,’’ *Journal of Object Technology*, August 2002,.
14. H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal, “Specifying subject-oriented composition,” in *TAPOS,* 2(3), 1996.
15. M. Robillard & G. Murphy, “Analyzing Concerns Using Class Member Dependencies,” Position paper for the ICSE Workshop on Advanced Separation of Concerns in SE, 2001.
16. H. A. Simon, *Models of bounded rationality*, vol. 2, Cambridge, MA (MIT Press, 1982).
17. T. Soued, N. Yahiaoui, L. Seinturier, B. Traverson, “Techniques d'aspect pour la gestion de la mémoire répartie dans un environnement CORBA/C++”, in *proc. of NOTERE'05,* 2005.
18. T. Reenskaugh, in *Working with Objects*, Prentice-Hall, 1995.
19. F. Steimann, “Domain Models are Aspect Free,” in *Proc of MODELS’05*, 2005.
20. F. Tip, J-D Choi, J. Field, and G. Ramalingam, “Slicing class hierarchies in C++”, In *Proc. of OOPSLA'96*, 1966.
21. P. Zave and M. Jackson, “Conjunction as Composition,” in *ACM Trans. on Softw. Eng. Methodol.*, vol. 2, no. 4, pp. 379-411, 1993.

Algebraic Specification of a Model Transformation Engine*

Artur Boronat, José Á. Carsí, and Isidro Ramos

Department of Information Systems and Computation,
Technical University of Valencia,
Camí de Vera s/n,
Valencia 46022, Spain
{aboronat, pcarsi, iramos}@dsic.upv.es

Abstract. In Model-Driven Engineering, a software development process is a sequence of manipulation tasks that are applied to models, where model transformations play a relevant role. MOMENT (MOdel manageMENT) is a framework that is integrated in the Eclipse platform. MOMENT provides a collection of generic set-oriented operators to manipulate EMF models. In this paper, we present the model transformation mechanism that is embodied by the *ModelGen* operator. This operator uses the term rewriting system Maude as transformation engine and provides support for traceability. *ModelGen* has been defined in an algebraic specification so that we can use formal tools to reason about transformation features, such as termination and confluence. Furthermore, its application to EMF models shows that formal methods can be applied to industrial modeling tools in an efficient way. Finally, we indicate how the *ModelGen* operator provides support for the QVT Relations language in the MOMENT Framework.

Keywords: Model-Driven Engineering, Model Transformation, QVT, Algebraic Specifications, Traceability.

1 Introduction

Nowadays, the Model-Driven Architecture (MDA) [1] and the Software Factories [2] initiatives are leading the Model-Driven Engineering field. Both agree that any software artifact in a software development process can be dealt with as a model. Models provide a more abstract description of a software artifact than the final code of the application. Therefore, working on models increases the productivity in a software development process. It also increases the portability and the quality of the final code by applying generative techniques. In MDA, model transformations have become a relevant issue by means of the forthcoming standard Query/Views/Transformations (QVT) [3]. Since any software artifact can be viewed as a model, model transformation is the basic mechanism that permits the manipulation of software artifacts [4].

* This work was supported by the Spanish Government under the National Program for Research, Development and Innovation, DYNAMICA Project TIC 2003-07804-C05-01.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 262 – 277, 2006.

Within this arena, the Model Management discipline [5] considers models as first-class citizens and provides a set of generic operators to manipulate them: *Merge*, *Diff*, *ModelGen*, etc. We have developed a framework, called MOMENT (MOdel manageMENT) [6], that is embedded in the Eclipse platform and that provides a set of generic operators to deal with models through the Eclipse Modeling Framework (EMF). An algebra of model management operators has been specified generically by using the Maude algebraic specification formalism [7] in the MOMENT framework.

In this paper, we focus on the *ModelGen* operator, the model transformation mechanism of MOMENT, which was presented as a proposal in [8]. This operator provides support for the QVT Relations language [3] and also provides support for traceability. *ModelGen* is used by the other model management operators of the Framework when a model manipulation has to be performed. Since the *ModelGen* operator is algebraically specified in Maude, this term rewriting system is used as the underlying runtime environment for model transformations in MOMENT. This fact provides an efficient environment to execute the *ModelGen* operator and a formal environment where algebraic features can be proved, such as the confluence and the termination of a model transformation.

The structure of the paper is as follows: Section 2 presents an overview of the QVT support in the MOMENT Framework and an example; Section 3 presents the algebraic specification of the *ModelGen* operator, how it is related to a QVT transformation, and the execution of a model transformation in the Framework; Section 4 compares our approach with other model transformation tools. Finally, Section 5 summarizes the main contributions of this paper.

2 Model Transformations in the MOMENT Framework

The *ModelGen* operator embodies the model transformation mechanism in the MOMENT Framework. This operator has been algebraically specified, although we use it to manipulate graphical models. To deal with models from an industrial standpoint and to manipulate them from a mathematical standpoint, we use two complementary Technical Spaces. A Technical Space (TS) is a working context with a set of concepts, a body of knowledge, tools, required skills, and possibilities [9]. We use the EMF and Maude technical spaces in our Framework. The former is characterized by its interoperability with industrial tools for solving actual Software Engineering problems. The latter constitutes the formal backbone of our model management approach.

Maude is a declarative language in the strict sense of the word. That is, a Maude program is a logical theory, and a Maude computation is logical deduction using the axioms specified in the theory/program. Maude is based on rewriting logic, which includes membership equational logic [19]. In Maude, membership equational theories are defined in functional modules. Computation is the form of equational deduction in which equations are used from left to right as simplification rules, with the rules being Church-Rosser and terminating.

Each Maude module specifies not just a theory, but also an intended mathematical model. For functional modules such models consist of certain sets of data and certain functions defined on such data, and are called algebras. Under Church-Rosser and

termination assumptions, the equations of a functional module evaluate algebraic expressions to a single final result. By definition, the results of operations in this algebra are exactly those given by the Maude interpreter. Thus, a Maude module can simultaneously be viewed as an executable formal specification and as a program.

To achieve the interoperability between the EMF and the Maude technical spaces, two kinds of bridges are used in our Framework: two for regular metamodels and one for the QVT Relations metamodel.

For regular metamodels, two bridges are defined between the two technical spaces, at the M2-layer and at the M1-layer (using the Meta-Object Facility [10] terminology). Both of these permit the integration of MOMENT with EMF. The projection mechanism at the M2-layer automatically obtains the algebraic specification[1] that corresponds to a regular metamodel, by applying generative programming techniques[2]. An algebraic specification that is generated in this way provides the constructors that are needed to define models of the corresponding metamodel as sets. The inverse projection mechanism that obtains an EMF metamodel from an algebraic specification is not relevant in our tool because the algebraic specification must conform to several features in order to be used by our operators and they should be automatically achieved. We also think that visual modeling environments are more suitable for defining these metamodels. At the M1-layer, we have developed a bidirectional projection mechanism that permits us to project an EMF model as a term of an algebra and to project a term of a metamodel algebra as an EMF model. In this case, bidirectionality is needed to apply an operator to an input model since the input model must be serialized as a term and the output term must be deserialized into an EMF model in order to be persisted.

Another bridge is defined for the QVT Relations metamodel. This permits the projection of a QVT transformation as an algebraic specification. This specification constitutes the axiomatic presentation of the *ModelGen* operator, when a transformation is defined among several metamodels.

In Fig. 1, the projection bridges between the two Technical Spaces (EMF and Maude) that are used in this work are shown. On the one hand, the dashed arrows represent the projection mechanism that obtains an algebraic specification for a regular EMF metamodel. On the other hand, the dotted arrow represents the projection mechanism that obtains the algebraic specification of a model transformation between several metamodels, when it is invoked in one direction. The importation of metamodels in an EMF QVT transformation model is also projected into the Maude TS by using the Maude module importation mechanism (shown as continuous arrows in the figure). Taking into account both kinds of projectors, we can deal with a QVT Relations model as the description of a model transformation or as a

[1] The algebraic specification that is generated for a given metamodel (defined in EMF as an Ecore model) permits the representation of models as algebraic terms. Thus, models can be manipulated by our model management operators. Algebraic specifications of this kind do not specify operational semantics for the concepts of the metamodel; they only permit the representation of information for model management issues.

[2] These bridges can also be formalized in MOMENT by considering the Maude metamodel. Then, the bridges can be defined as model transformations.

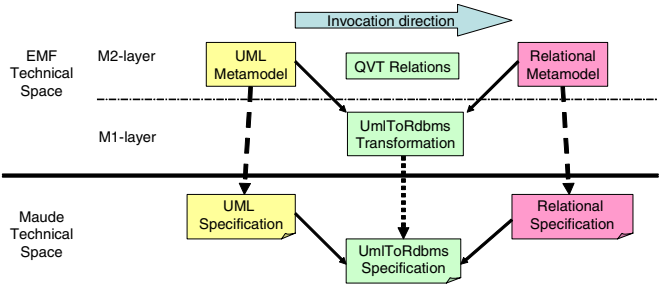


Fig. 1. Overview of the QVT support in the MOMENT Framework

regular model. Therefore, it can be transformed by using another QVT Relations model. This fact permits us to define the *ModelGen* operator as a mechanism to obtain higher order transformations through the QVT Relations language.

We have chosen the UmlToRdbms transformation that is presented in the QVT final adopted specification [3] as an example to illustrate the use of the *ModelGen* operator in the MOMENT Framework[3]. The Ecore metamodel [11] has been used as implementation of the UML metamodel. The RelationalDMBS metamodel of the QVT proposal has been specified as an EMF metamodel. Using both metamodels, the UmlToRdbms transformation is applied to the source UML model in Fig. 2 to obtain the target relational schema, which is shown in the figure by using the default EMF graphical modeller.

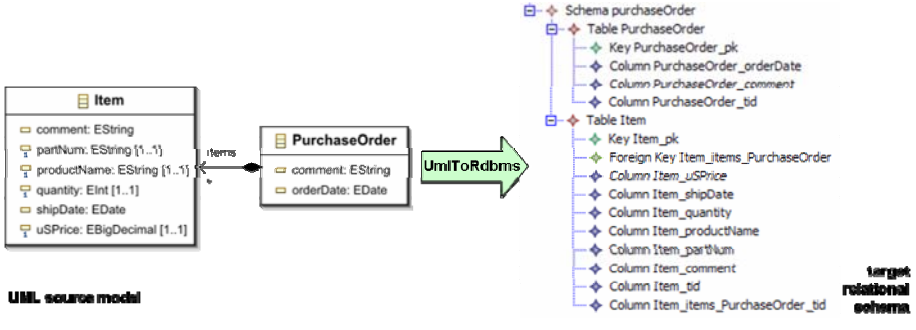


Fig. 2. Example of transformation of a UML model into a relational schema

3 The Model Transformation Mechanism in MOMENT: The *ModelGen* Operator

In this section, an overview of the QVT Relations language support in MOMENT that is based on the *ModelGen* operator is provided. We indicate how the model transformation in Section 2 is specified with the *ModelGen* operator, taking the Relations language as reference point.

[3] The complete algebraic specification of the example can be found at [25].

3.1 Overview of the Model Transformation Mechanism

In the QVT Relations language, a model transformation is defined among several metamodels, which are called the domains of the transformation. A QVT transformation is constituted by QVT relations, which become declarative transformation rules. A QVT relation specifies a relationship that must hold between the model elements of different candidate models. The direction of the transformation is defined when it is invoked by choosing a specific domain as target. If the target domain is defined in the QVT transformation as enforceable, a transformation is performed. If the target domain is defined as checkonly, just a checking is performed.

In MOMENT, a QVT transformation is defined by means of the ModelGen operator. QVT relations are defined by means of the ModelGenRule operator[4], which is used by the former operator. A model transformation can be applied to several source models, which may or may not conform to the same metamodel. It generates one target model and a set of traceability models. A traceability model contains a set of traces that relate the elements of the source model to the elements of the target model, indicating which transformation rule has been applied to each source element.

In this section, we present how MOMENT executes the *ModelGen* operator, transforming the UML model of the example in Section 2 into a relational schema. Fig. 3 shows the two MOF layers involved in a model transformation: the M2-layer, where the metamodels are defined; and the M1-layer, where the model transformation and the models are defined and manipulated. The front part of the figure represents the front-end of the MOMENT framework, i.e. EMF and all the plug-ins that are built on it. The back part of the figure represents the formal back-end of the MOMENT Framework, where Maude remains. The traceability support has not been taken into account in the figure.

Fig. 3 represents the transformation of the UML model by using the *ModelGen* operator. The steps that are automatically performed by the MOMENT Framework when the *ModelGen* operator is applied to the source UML model are the following:

- *(1) and (2):* We specify both UML and RelationalDMBS metamodels at the M2-layer by means of the EMF or graphical editors based on this modelling framework. For instance, we can also consider XML schemas and Rational Rose models as metamodels.
- *(3)*: The QVT transformation is defined as a model at the M1-layer, but it relates the constructs of the source to the constructs of the target metamodels. The transformation has to be defined as a model that conforms the QVT Relations metamodel by means of a graphical interface or as a program using the Relations language. The transformation model can either be defined by the user or automatically produced by another transformation.
- *(4)*: We define a UML model using a UML graphical editor based on EMF.
- *(5) and (6)*: Both UML and RelationalDBMS metamodels, respectively, are projected as algebraic specifications by means of the interoperability bridges that

[4] In the MOMENT framework, ModelGen and ModelGenRule axioms are just generated when the target domain of the invoked transformation is defined as enforceable. The algebraic support for checkings is provided by means of boolean operators and it is out of the scope of the paper.

have been implemented in the MOMENT Framework. These algebraic specifications permit each metamodel to be considered as an algebra that provides the constructors to build models as algebraic sets.

- *(7)*: The model that defines the QVT transformation is projected into the Maude TS as the UmlToRdbms algebraic specification, which contains the specification of the *ModelGen* and *ModelGenRule* operators.
- *(8)*: The source UML model, which is defined in step *(4)* at the M1-layer, is projected into the Maude TS as a term of the UML algebra *(9)*.
- *(10)*: Maude applies the *ModelGen* operator through its equational deduction mechanism, obtaining a term of the RelationalDBMS algebra (11). Thus, Maude constitutes the runtime engine for the MOMENT transformation mechanism.
- *(12)*: This is the last step of the model transformation process. It parses the term *(11)*, defining an EMF model *(13)* in the M1-layer, which conforms to the target metamodel defined at the M2-layer.

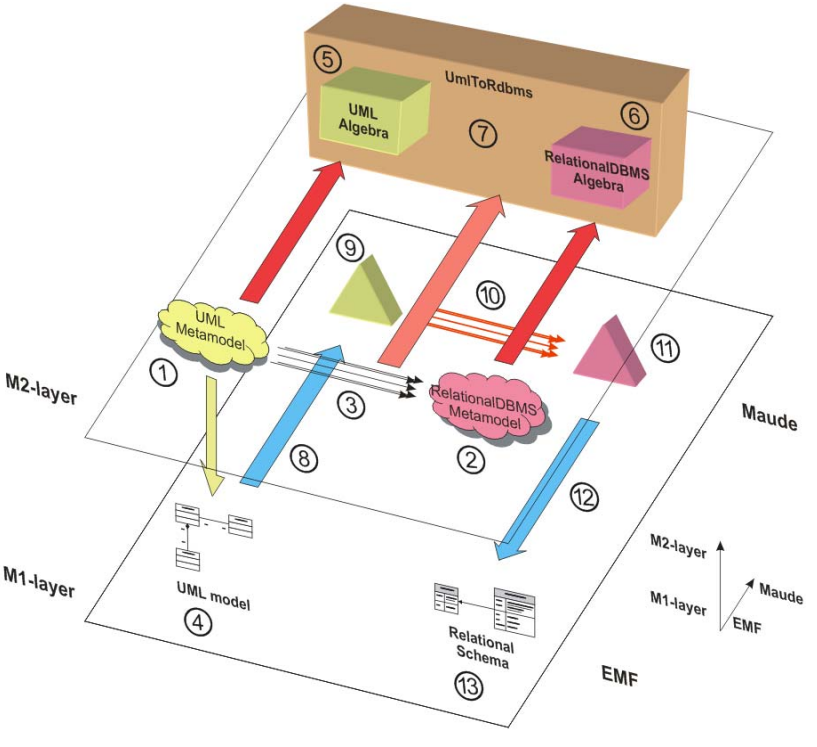


Fig. 3. Steps performed by the MOMENT model transformation mechanism

In the model transformation process, the user only interacts with the MOMENT framework when defining the source and target metamodels (step (1) and (2)), the QVT transformation between both metamodels (step (3)) and the source model (step (4)). The other steps are automatically carried out by the framework. The output model can also be manually manipulated from a graphical editor. In this paper, we

focus on the steps (7) and (10), indicating the structure of the *ModelGen* operator and how it is related to the QVT Relations metamodel.

3.2 Relations: The *ModelGenRule* Operator

In MOMENT, a QVT transformation is projected into several directed transformations, by means of ModelGen operator specifications. There is exactly one transformation for each direction in which the QVT transformation can be invoked. QVT relations of a QVT transformation are projected as ModelGenRule axioms, once the transformation direction is determined. The operation *ModelGenRule* that permits the definition of directed transformation rules is declared as follows:

```
op ModelGenRule`(_;_`) : RelationSymbol ParameterList ->
    Tuple{ TTargetMM, TTraceabilityMetamodel,...} .
```

The *ModelGenRule* operator has two parameters: the name of the relation (a term of the sort RelationSymbol) and a polymorphic list of parameters. The membership equational logic [7] is used to define the operational semantics of this operator by means of equations. To use the Maude equational deduction as the runtime engine for our transformation rules, we need them to be confluent and terminating. The first parameter makes the confluence satisfaction easier to achieve. It permits to differentiate two transformation rules, even though they have the same list of parameters. Therefore, we avoid the situation where several equations can be applied to reduce the same term. We discuss the termination issue in more depth in Section 3.2.2. The second parameter permits the definition of a polymorphic list of parameters for the transformation rule. This means that we can define a parameter of any type (either a model or a basic type) as input for the transformation rule.

The result of a transformation rule is a tuple of several elements, where an element can be a model conforming any of the metamodels involved in the transformation. Among the models that are produced by a transformation rule, we distinguish the target model and several traceability models. There is one traceability model for each pair *(source model, target model)*.

3.2.1 Structure of a Transformation Rule in MOMENT

We can only know the direction of a transformation by means of the enforceable property during the invocation process. During this process, the *ModelGenRule* axioms that specify the operational semantics of the relation are generated to be invoked later. Fig. 4 shows the two parts that constitute a *ModelGenRule* axiom: the specification of the model transformation and the specification of the traceability model definition. The Maude code that is generated for each part of a QVT relation is structured as shown in Fig. 4.

- The Relation Symbol. This is the name of the transformation rule. It is used to define a constructor for the sort *RelationSymbol*, which is used in the *ModelGenRule* axioms for the sake of confluence.
- The Domains. They constitute the patterns of the transformation rules and the body, using constructs of the involved metamodels. Two kinds of domains can be distinguished in the transformation invocation:

- The domain that is selected as target, when the transformation is invoked, is used to generate the body of the transformation rules. In the body of a transformation rule, new instances can be created in the target model and expressions to obtain the information from the source models can be used. Expressions of this kind are called object template expressions. In them, OCL is used as the query language and new functions can be declared in a transformation to manipulate data. On the one hand, a parametric algebraic specification has been defined to provide the operational semantics for OCL 2.0 expressions[5] in MOMENT. On the other hand, Maude itself is suitable to define the operational semantics of QVT functions.
- The source domains of the QVT relation are used to define the pattern of the transformation rule. In the pattern, each input model appears twice. This is needed to achieve two goals at the same time: to use the pattern matching mechanism that Maude provides by means of recursion and to provide support for OCL expressions. The first model is used to search the corresponding element of the pattern that is needed in the transformation rule. The second model is needed because Maude does not provide support for side effects, as in pure functional programming languages. This forces us to keep the whole model throughout the term reduction process in order to navigate it by means of OCL expressions, which can be used in the guard of the pattern. In the following section, we study the pattern matching mechanism in further detail.

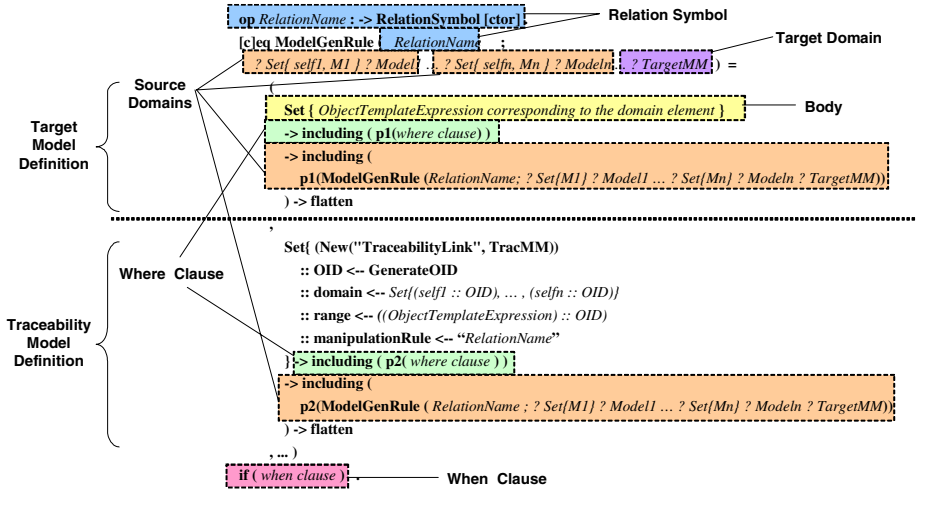


Fig. 4. Axiom for the *ModelGenRule* operator

– The *When clause.* This clause is used as a precondition for the transformation rule. Therefore, this guard participates in the pattern matching mechanism. It is used to obtain the condition of the equations that constitute the transformation rule. The

[5] The OCL 2.0 algebraic support is out of the scope of this paper.

resulting equations are applied by the conditional pattern matching mechanism of Maude.

– The *Where clause*: this clause is used as postcondition for a transformation rule. In MOMENT, it is used to generate the code that invokes other QVT relations. In this clause, variables can also be initialized with new values to be used in the transformation.

In the QVT proposal, the traceability support is implicit in the QVT Relations language and is explicit in the QVT Core language. The *ModelGen* operator also generates traceability models automatically, but its definition has to be explicitly specified in the *ModelGenRule* axioms. The traceability model produced by the *ModelGen* operator conforms to the traceability metamodel that is presented in [12]. Thus, it can also be manipulated as just another model by the operator *ModelGen*.

In the second part of the axiom shown in Fig. 4, a new trace (instance of the TraceabilityLink class) is added to the traceability metamodel. Its *domain* field is filled with the identifiers (or references) of the source elements that have been matched with the pattern. Its *range* field is filled with the identifiers of all the instances that are created in the body of the transformation rule. Finally, the *manipulationRule* field indicates which transformation rule has been applied. Since the axioms for the *ModelGenRule* transformation rules are automatically generated from a QVT relations model, the traceability support in MOMENT is automatically generated and is kept hidden from the final user.

As an example of a *ModelGenRule* axiom, we show the axiom that represents the regular case pattern for the following relation *PackageToSchema of the UmlToRdbms transformation* (when the transformation is applied to a UML model):

QVT Relation:

```
top relation PackageToSchema // map each package to a schema
{    pn: String;
     checkonly domain uml p:Package {name=pn};
     enforce domain rdbms s:Schema {name=pn};
}
```

Maude Axiom:

```
ceq ModelGenRule (PackageToSchema ; ? Set{ self, M } ? Model ? TargetMM) =
(    Set{ (New("Schema", TargetMM))
           :: OID <-- (self :: OID)
           :: name <-- (self :: name)
     } -> including ( p1(ModelGenRule (PackageToSchema ; ? Set{ M } ? Model ? TargetMM))) -> flatten
,    Set{ (New("TraceabilityLink", TracMM))
           :: OID <-- generateOID
           :: domain <-- Set{ (self :: OID) }
           :: range <-- Set{ (self :: OID) }
           :: manipulationRule <-- "EPackage-to-Schema"
     } -> including( p2(ModelGenRule (PackageToSchema ; ? Set{ M } ? Model ? TargetMM)) ) -> flatten
) if (self :: ecore-EPackage) .
```

3.2.2 Pattern Matching

The pattern matching mechanism that is used to apply the transformation rules to source models is provided by Maude. To understand how the *ModelGenRule* uses it, we study the parts of a transformation rule that are involved in the pattern matching: the *source domains* and the *when clause*.

A source domain is a model used as input in a transformation. It is a set that is constituted by an associative commutative magma[6] of elements. The generic constructors for a Set are the following:

```
op _,_ : Magma{X} Magma{X} -> Magma{X} [assoc comm ctor] .
op Set{_} : Magma{X} -> Set{X} [ctor] .
```

Taking into account that the sort that represents model elements is a subsort of *Magma{X}*, a UML model that is constituted by a package and a class can be defined as a set with the following term: *Set{(UML-Package …) , (UML-Class …) }*; where *UML-Package* and *UML-Class* are the constructors that correspond to these concepts in the Maude projection of the UML metamodel. By using a recursion mechanism we can use the Maude pattern matching mechanism. We define the following variables: the variable $self_i$ can be bound to a term that represents any element of a $Model_i$ model. The variable M_i can be bound to any magma of elements of a $Model_i$ model. For each QVT relation, we obtain two axioms for the *ModelGenRule* operator by using the following patterns:

– The base case *Set{ $self_i$ }*: the right-hand side of this equation is constituted by the tuple of the target model and the traceability models.

```
eq ModelGenRule (RelationName ; ? Set{ self_i } ? Model_i … ? TargetMM) = (Set{…}, Set{…},…) .
```

– The regular case *Set{ $self_i$, M_i }*: the right-hand side of this equation is constituted by the tuple of the target model and the traceability models, but also contains the application of the recursive transformation rule *RelationName* to the models that are constituted by the residuary magmas of elements. In the first element of the tuple, we apply the recursive operation and we add the first argument of the returning tuple by means of *p1*. In the second element, we use *p2* to get the second element of the returning tuple.

```
eq ModelGenRule (RelationName ; ? Set{ self_i, M_i } ? Model … ? TargetMM) =
(Set{…}->including(p1(ModelGenRule (PackageToSchema ; ? Set{ M_i } ? Model_i ? TargetMM)))->flatten
,
Set{…}->including(p2(ModelGenRule(PackageToSchema ; ? Set{ M_i } ? Model_i ? TargetMM)) )->flatten).
```

The *when clause* also provides relevant information to determine whether or not an axiom can be matched to perform a transformation. When the guard described in the *when clause* does not contain an OCL query that searches elements through a model, it can be added to the axiom as a condition, turning the axiom into a conditional equation. When the guard contains an OCL query, the guard is added as an *if…then…else…fi* clause in the body of the equation. Both considerations are needed for the sake of the efficiency of the Maude AC matching algorithm.

When no axiom of the *ModelGenRule* operator can be applied, there is a general axiom for all the transformation rules that is applied by default. This axiom guarantees the termination of a transformation and is specified as follows:

```
var TR : RelationSymbol .      var PL : ParameterList .
eq ModelGenRule (TR ; PL) = (empty-set, empty-set ) [owise] .
```

[6] A magma of elements represents a group of elements that may be encapsulated in a set.

3.3 Transformations: The *ModelGen* Operator

The *ModelGen* operator provides the model transformation mechanism in the MOMENT Framework. This operator corresponds to a QVT transformation when it is invoked in a specific direction. The *ModelGen* operator is declared as follows:

```
op ModelGen`(_;_`) : TransformationSymbol ParameterList
-> Tuple{ TTargetMM, TTraceabilityMetamodel,... } .
```

Similarly to the *ModelGenRule* operator, it has two formal parameters: the symbol that represents the name of the transformation and a polymorphic list of parameters for the transformation. The result of the operator is a tuple that is constituted by the resulting target model and by traceability models. There is one traceability model for each pair *(source model, target model)*.

The code that is generated for a transformation includes the definition of the symbol of the transformation and an axiom that specifies the semantics of the transformation. Fig. 5 shows the structure of an axiom for the *ModelGen* operator. With regard to the definition of a QVT transformation, we identify the following parts:

- The transformation symbol: represents the name of the transformation.
- The parameters of the transformation:
  - Source Domains. *? SourceModel1 … ? SourceModeln* constitute a list of terms that represent the source models of the transformation.
  - Target Domain. *TargetMM* is a term that represents the target metamodel. It is used to create new instances of the classes that appear in the metamodel by means of the Maude reflection mechanism.
  - List of names of the input and target models. This list is used to define traceability models. This information is needed to indicate the models that are related by means of a traceability model.
- Target model definition. This is the first argument of the resulting tuple, where the target model is generated by applying the *ModelGenRule* axioms to the elements of the source models.
- Traceability model definition. The other arguments of the resulting tuple are the traceability models that relate the elements of the target model to those of each source model. Furthermore, a new instance of the *TraceabilityModel* class is created to relate a source model to the target one.

The operator *CompleteReferences* is needed to simplify the definition of transformation rules when the metamodel has opposite references. In EMF, UML associations that may appear in a MOF metamodel are defined by means of opposite references (two instances of the *EReference* class) [11]. If the metamodel has opposite references, both references must be initialized when a model is defined. This fact can make the definition of transformations more complex. Although the EMF editor solves this problem automatically, an independent solution is needed. The *CompleteReferences* operator fulfils this goal. Thus, it permits MOMENT to remain independent of any other technological space that is different from the Maude TS and to compose several transformations without loss of information. This operator has two

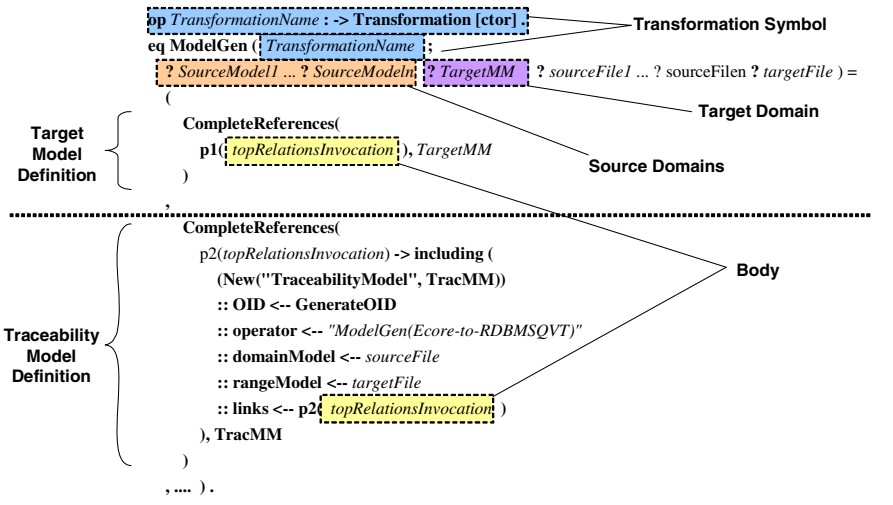


Fig. 5. Axiom for the *ModelGen* operator

input parameters: the model to be completed with references and the corresponding metamodel, which is expressed as a model conforming the Ecore metamodel. The *CompleteReferences* operator uses the Maude reflection mechanism to traverse the whole input model in order to complete the forgotten references. The output of the operator is the completed model.

Taking into account the example of the UmlToRdbms transformation that is presented in the MOF QVT final specification, we indicate the Maude code that provides the semantics of the transformation when it is invoked to transform a UML model into a relational schema. In the *ModelGen* axiom, we have only added the code to obtain the first element of the returning tuple, i.e. the target model.

QVT Transformation:
transformation umlToRdbms(uml:SimpleUML, rdbms:SimpleRDBMS) {
top relation ClassToTable {…}
top relation PackageToSchema {…}
top relation AssocToFKey {…}
… }
Maude code:
op umlToRdbms : -> Transformation [ctor] .
***eq** ModelGen (umlToRdbms; ? Model ? TargetMM ? sourceFile ? targetFile) =*
(CompleteReferences(
p1(ModelGenRule (PackageToSchema ; ? Model ? Model ? TargetMM))
-> including(p1(ModelGenRule (ClassToTable ; ? Model ? Model ? TargetMM)))
-> including(p1(ModelGenRule (AssocToFKey ; ? Model ? Model ? TargetMM))) -> flatten
, TargetMM), …) .

4 Related Work

We provide a brief comparison of the ModelGen operator to other model transformation mechanisms that are the most current in the Model-Driven Engineering field [20].

The goal of the study is to compare their support for traceability and transformation organization.

MTF [13] is the IBM Model Transformation Framework, which implements some of the QVT concepts and is based on the EMF. It provides a simple declarative language for defining mappings between models. An MTF transformation results in a set of mappings that relate objects among different models. The direction of the transformation is defined when the transformation is invoked. Bidirectional transformations imply constraining the kind of model transformations that can be solved. For instance, transformations that produce loss of information cannot be taken into account.

ATL [14] is a model transformation language that provides declarative and imperative constructs. A transformation is constituted by several transformation rules but there is no mechanism to organize them by means of modules. Traceability support is not provided implicitly in a transformation; traceability models can be generated by a transformation as in the ModelGen operator. The expressions in a transformation rule are defined in OCL expressions, making the ATL language easy to learn and to use. Tefkat [15] is a model transformation tool that is quite similar to ATL, which is also built on EMF. It incorporates the concept of tracking classes to define traces between the source model and the target model that is generated by the transformation. However, traces must be defined explicitly.

XSLT has become a popular alternative for describing model transformations. Tools, such as MTRANS [16] or UMT [17], follow this approach by serializing metamodels and models into XMI documents and then performing the transformations by means of XSLT specifications. Nevertheless, the verbosity of the XML syntax sometimes leads to specifications that are difficult to read and to maintain [18]. XSLT 2 permits the definition of transformations that generate more than one XML document. Therefore, traceability support can be added explicitly to a XSLT transformation. This approach can also be applied to EMF models, which are persisted in XMI. EMF avoids the serialization of certain data, such as derived attributes or default values. Thus, this functionality must be embedded in the XSLT specification. This task can easily become cumbersome.

In [21], graph transformation technology is studied by applying a taxonomy. Graph-based model transformations usually take advantage of the visual nature of the graphs to specify transformation rules. Composition of graph transformations can be achieved by using controlled or programmed graph transformation, such as sequencing, branching or looping. For instance, Fujaba [22] uses story diagrams for this purpose, while VIATRA [23] uses abstract state machines. In Fujaba, transformations are implemented as method bodies, so composition of transformations can be achieved by performing method calls. In MOMENT, transformations are defined in algebraic modules that can be imported into others. Furthermore, the ModelGen operator can be easily composed with other operators by using functional composition since we are working in an algebraic context. This feature avoids the development of complex frameworks to deal with transformation composition. Since the algebraic definition of the ModelGen operator is automatically generated from a QVT model, MOMENT can use standard-based notation, graphical or textual, to specify transformations.

Graph-based model transformation technology has a formal background where mathematical features such as confluence and termination can be proved. For

instance, AGG provides the mechanism of critical pair analysis to check termination and confluence of graph grammars [24]. As a Maude module is by construction a mathematical object, we can directly use all the tools of mathematics and logic, including automatic or semiautomatic tools, to reason about the correctness of Maude modules. The Maude interpreter itself is the first and most obvious such tool. In fact it is a high-performance logical engine to prove logical facts about our theories. Furthermore, Maude has a collection of formal tools supporting different forms of logical reasoning to verify program properties that can be directly applied to the algebraic specifications obtained in MOMENT, including: an inductive theorem prover (ITP) to verify properties of functional modules; a Church-Rosser checker, to check such a property; a Knuth-Bendix completion tool and termination checker.

Among the studied tools, MTF is the only one that provides implicit traceability support. In the other approaches, the traceability support must be explicitly codified in the transformation function to generate a traceability model. In MOMENT, the set of axioms that describe the ModelGen operator is automatically generated from a QVT Relations model. Thus, the traceability support is also implicit.

5 Conclusions

In this paper, we have presented the ModelGen operator, which permits the definition of directed declarative transformations that can be applied to several source models, which may or may not conform to the same metamodel. Furthermore, a transformation can be parameterized with additional data that can act as control parameters allowing configuration and tuning. The return value of a transformation is a tuple that is constituted by a model and several traceability models. There is one traceability model for each pair *(source model, target model)*. The traceability support that is provided in MOMENT [12] permits the definition of an incremental transformation operator in an easy way, similar to the PropagateChanges operator, which is also presented in [12].

Our formal approach for model transformation permits the application of advantageous features to this field, such as transformation composition or modularity. Furthermore, formal features (like confluence and termination) can also be studied. The MOMENT Framework shows that formal methods can be applied to industrial tools not only for proving theoretical aspects, but also for solving actual problems in an efficient manner. Nevertheless, an algebraic setting might not be the most user-friendly environment to work on models. This reason led us to provide support for QVT by using generative programming techniques. In this way, Maude remains hidden from the final user although its formal features are used in our framework.

References

1. Frankel, D. S.: Model Driven Architecture: Applying MDA to Enterprise Computing. John Wiley & Sons OMG Press. January, 2003.
2. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley & Sons. 2004.

3. OMG, "MOF 2.0 QVT final adopted specification (ptc/05-11-01)", November 2005.
4. Sendall, S., Kozaczynski, W. Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software. Sep/Oct 2003 (Vol. 20, No. 5), pp. 42-45.
5. Bernstein, P.A: Applying Model Management to Classical Meta Data Problems. pp. 209-220, CIDR 2003.
6. The MOMENT web site: http://moment.dsic.upv.es:8080
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theoretical Computer Science, 285(2):187-243, 2002.
8. Boronat, A., Carsí, J.Á., Ramos, I.: Automatic Reengineering in MDA Using Rewriting Logic as Transformation Engine. IEEE Computer Society Press. 9th European Conference on Software Maintenance and Reengineering. Manchester, UK. 2005.
9. Kurtev, I., Bézivin, J., Aksit, M.: Technological Spaces: An Initial Appraisal. Int. Federated Conf. (DOA, ODBASE, CoopIS), Industrial track, Irvine, 2002.
10. OMG, "Meta Object Facility 1.4", http://www.omg.org/technology/documents/formal/mof.htm
11. EMF web site: http://www.eclipse.org/emf/
12. Boronat, A., Carsí, J.Á., Ramos, I.: Automatic Support for Traceability in a Generic Model Management Framework. LNCS 3748. In Proceedings of European Conference on Model Driven Architecture - Foundations and Applications. Nuremberg (Germany). 2005.
13. The MTF web site: http://www.alphaworks.ibm.com/tech/mtf
14. Bézivin, J., Dupé, G., Jouault, F. , Pitette, G., and Rougui, E.J.: First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In: OOPSLA 2003 Workshop, Anaheim, California.
15. Lawley, M., Steel, J.: Practical declarative model transformation with tefkat. In Model Transformations In Practice Workshop, Montego Bay, Jamaica, October 2005.
16. Peltier, M., Bézevin, J., Guillaume, G.: MTRANS: A general framework, based on XSLT for model transformations. In WTUML '01, Proceedings of the workshop on Transformations in UML, Genova, Italy, 2001.
17. The UMT web site: umt-qvt.sourceforge.net/
18. Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The Missing Link of MDA, In A. Corradini, H. Ehrig, H.-J. Kreowski, G. Rozenberg (Eds.): Graph Transformation: First International Conference (ICGT 2002), v1 15 Barcelona, Spain, October 7-12, 2002. Proceedings. LNCS vol. 2505, Springer-Verlag, 2002, pp. 90 – 105
19. Meseguer, J.: Membership algebra as a logical framework for equational specification. In Francesco Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3-7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18-61. Springer, 1998.
20. Czarnecki, K., Helsen, S.: Classification of Model Transformation Approaches. OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture.
21. Mens, T., Van Gorp, P., Varro, F., Karsai, G.: Applying a model transformation taxonomy to graph transformation technology. Proc. Int'l Workshop on Graph and Model Transformation (GraMoT 2005). September 2005.
22. Burmester, S., Giese, H., Niere, J., Tichy, M., Wadsack, J., Wagner, R., Wendehals, L., Zuendorf, A.: Tool integration at the meta-model level: The fujaba approach. Int'l Journal on Software Tools for Technology Transfer 6 (2004), pp. 303–318.

23. Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: VIATRA - visual automated transformations for formal verification and validation of UML models, in: Proc. 17th Int'l Conf. Automated Software Engineering (2002), pp. 267–270.
24. Heckel, R., Küster, J., Taentzer, G.: Confluence of typed attributed graph transformation systems, in: Proc. 1st Int'l Conf. Graph Transformation, LNCS 2505 (2002), pp. 161–176.
25. Algebraic specification of the model transformation that is used as example in the paper: http://moment.dsic.upv.es/publications/fase06/example.html

Fundamentals of Debugging Using a Resolution Calculus*

Daniel Köb and Franz Wotawa

Technische Universität Graz,
Institute for Software Technology,
8010 Graz, Inffeldgasse 16b/2, Austria
{koeb, wotawa}@ist.tugraz.at
http://www.ist.tugraz.at/

Abstract. Detecting and localizing a fault within a program is a non-trivial and time consuming task. Most of the efforts spent for automating the task have focused on fault detection. In this paper we shift the focus on fault localization. We introduce a resolution calculus that allows for representing the program's behavior based on correctness assumptions. The fault localization task is reduced to finding consistent assumptions which are represented as a non-monotonic reasoning process where efficient algorithms exist. Finally, we compare our approach with a previous approach to fault localization that is based on trace analysis. As a result we can show that our approach is less sensitive to search assumptions.

1 Introduction

Detecting, locating, and correcting bugs in programs are time consuming and expensive tasks and are summarized under the term debugging. Currently, debugging is hardly automated with some rare exceptions. For example in fault detection, tools for automated verification and test case generation have been available. However, there have been almost no tools and techniques for automated fault localization presented so far. In this paper we introduce foundations for automated debugging which are based on the program's semantics. The underlying idea is to directly use the program's behavior for a given test case to derive the location of a bug. For this purpose we represent the behavior of a program in terms of a calculus and the test cases as predicates at certain lines in the code. Moreover, we make underlying assumptions about the state of statements explicit. This state can be either bug-free (not abnormal) or faulty (abnormal). These explicit assumptions are used to modify the program's behavior and to check the behavior against the given test case. For example, if we assume a statement to be faulty and all others to be correct, then we might not be able anymore to derive a value for a variable at a certain position in the code which contradicts a given test case. Hence, we know that the assumption is consistent

* The work presented in this paper has been supported by the Austrian Science Fund (FWF) grants P15265-INF and P17963-INF. Authors are listed in alphabetical order.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 278–292, 2006.

with the given test case and as a consequence that assuming the statement to be faulty explains differences between the intended behavior given by the test case and the detected behavior given by the program run. Because the semantics of a program, i.e., a model of the program's behavior, is directly used to identify the fault location, our approach is called model-based debugging. We illustrate the basic idea behind model-based debugging using the following tiny program, which implements the computation of the circumference `c` and the area `a` of a circle from a given radius `r`.

```
1. d = 2 * r;
2. c = d * 3.14;
3. a = d * d * 3.14;
```

Obviously, the program contains a bug in line 3. We further assume `r` to be `1` before line 1, and `c` to be `6.28` and `a` to be `3.14` after line 3. This test case can be represented in terms of predicates for certain positions in the code. For our example, we have the following situation:

$\{r = 1\}$
1. `d = 2 * r;`
2. `c = d * 3.14;`
3. `a = d * d * 3.14;`

$\{c = 6.28 \wedge a = 3.14\}$

We apply Hoare's calculus [8] to propagate predicates through the program. Propagating predicates forwards is always possible by applying a statement's semantics, whereas backwards propagation of predicates is not possible for all statements. Especially heap manipulating statements that produce garbage prevent backwards propagation. In particular we use only the rules for program statements and assignments:

$$\{P(e/x)\}\ x\ \texttt{=}\ e\ \{P\} \qquad \frac{\{P\}\ s_1\ \{Q\};\{Q\}\ s_2\ \{R\}}{\{P\}\ s_1\ \texttt{;}\ s_2\ \ \{R\}}$$

It is worth noting that these rules can only be used when assuming statements to be bug-free. Because if a statement is assumed to be faulty, we do not know anything about its correct behavior. But even with limited backwards propagation of predicates we are able to derive worthwhile information for debugging. Using these rules and assuming all statements to be correct we obtain the following logical sentences:

$\{r = 1 \wedge 6.28 = 2 \cdot r \cdot 3.14 \wedge 3.14 = 2 \cdot r \cdot 2 \cdot r \cdot 3.14\}$
1. `d = 2 * r;`

$\{6.28 = d \cdot 3.14 \wedge 3.14 = d \cdot d \cdot 3.14\}$
2. `c = d * 3.14;`

$\{c = 6.28 \wedge 3.14 = d \cdot d \cdot 3.14\}$
3. `a = d * d * 3.14;`

$\{c = 6.28 \wedge a = 3.14\}$

Obviously the sentences before line 1 and between lines 1 and 2 are unsatisfiable because it is not possible to derive different values for the same variables

at a specific position in the code. Because of the detected inconsistency the underlying assumption of all statements being bug-free cannot be correct. Hence, we have to change the assumptions. If we assume statement 1 to be faulty, we still derive an inconsistency. Moreover, when assuming a fault in statement 2, we again come to an unsatisfiable logical sentence. The argumentation is as follows:

From $r = 1$ and statement 1 we derive $d = 2$. From the given predicates at the end of the program we derive predicates before line 3.

$\{r = 1 \wedge 6.28 = 2 \cdot r \cdot 3.14 \wedge 3.14 = 2 \cdot r \cdot 2 \cdot r \cdot 3.14\}$

1. `d = 2 * r;`
   $\{r = 1 \wedge d = 2\}$
2. `[ c = d * 3.14; ] // ASSUMPTION: STATEMENT IS FAULTY`
   $\{c = 6.28 \wedge 3.14 = d \cdot d \cdot 3.14\}$
3. `a = d * d * 3.14;`
   $\{c = 6.28 \wedge a = 3.14\}$

However, we do not know anything about the behavior of the statement in line 2. To allow for deriving new predicates before and after the statement, we introduce a new assumption saying that Hoare's rules can be applied to predicates that do not contain the target variable of a statement that is assumed to be faulty. If using this assumptions, we finally again obtain an inconsistency and we know that statement 2 is no explanation for the misbehavior.

In summary we conclude that the correctness assumptions for statements 1,3 and 2,3 both lead to an inconsistency. Such assumptions that lead to inconsistencies are called conflicts. In order to remove both conflicts, we have either to assume statements 1,2 or statement 3 to be incorrect and the rest of the statements to be correct. Both of the statement sets are explanations to the misbehavior but the assumption that statement 3 contains a bug is the smallest explanation. Hence, using the semantics of the statements and a test case allows to compute a diagnosis, i.e., statements that when assumed to contain a bug explain differences between the program's behavior and the given test case.

The objectives of the paper are: (1) to introduce a formal model for program debugging that is based on well known paradigms, (2) to present the formal properties that follow from the given definitions, (3) to compare the outcome of the proposed methodology with other debugging techniques, and (4) to discuss limitations and related research.

The paper is organized as follows. We first introduce the basic definitions of model-based debugging. In the following section we briefly introduce the basic definitions of another approach to fault localization which is based on program traces. Afterwards we present a case study which compares the outcome of both approaches. Finally, we discuss related research and conclude the paper.

2 Model-Based Debugging

In the introduction we used an example to illustrate the basic idea behind model-based debugging (MBD). It is important for MBD to have a logical program model and a test case. The model in our case is a calculus that is based on

Hoare's calculus with some slight modifications. The modifications are due to the assumptions about the state of a statement to be either faulty or bug-free. These explicit assumptions are used to compute diagnoses. In the first part of this section we discuss the logical model for debugging. The second part is devoted to the definition of debugging and some of its consequences. The underlying definitions except the modeling of the program's behavior originate from model-based diagnosis [9] but are adapted to closely represent the fault localization within programs instead of physical systems.

The first step in providing the formal foundations of debugging is to state the debugging problem which has to be solved. As explained within the elaboration of our introductory example, we have to know about the program's behavior and about a test case containing the expected variable values at certain positions in the code.

Definition 1 (Debugging problem). *Given a program Π. A debugging problem is a tuple $(\mathcal{D}, \mathcal{T})$ where $\mathcal{D}$ is a set of logical rules, i.e., a calculus, that represents the behavior of the individual statements of Π, and $\mathcal{T}$ is a set of predicates which state the values of variables at certain positions in program Π.*

The model $\mathcal{D}$ of a given program Π in the definition of a debugging problem has to fulfill some requirements. First, the model has to provide means for reasoning about predicates which state the values of variables. Second, it is crucial that the model makes assumptions about the assumed state of a statement explicit. Whenever a statement is assumed to behave correct or incorrect the respective behavior has to be available within the model. Finally, reasoning should not only be from the beginning of a program to its end but also in the opposite direction.

Figure 1 shows a model for a simple imperative language. The $Ab\backslash 1$ predicate is used to represent the explicit assumption about the state of a statement, i.e., to be either faulty or bug-free. The latter is represented by the negation of the Ab predicate. For simplicity, the model further assumes that every statement of program Π has a unique corresponding index and that reasoning can be done in both directions if not otherwise stated.

$$\neg Ab(i) \Rightarrow \{P(e/x)\}\ x\ \texttt{=}\ e\ \{P\} \tag{1}$$

$$\frac{\{P\}\ s_1\ \{Q\}; \{Q\}\ s_2\ \{R\}}{\{P\}\ s_1\ \texttt{;}\ s_2\ \{R\}} \tag{2}$$

$$\neg Ab(i) \Rightarrow \frac{\{P \wedge p\}\ s_1\ \{Q\}; \{P \wedge \neg p\}\ s_2\ \{Q\}}{\{P\}\ \texttt{if}\ p\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end if}\ \{Q\}} \tag{3}$$

$$Ab(i) \Rightarrow \{P\}\ s\ \{P\} \text{ if } P \text{ does not contain any defined variable of statement } s \tag{4}$$

Fig. 1. Meta-model of debugging calculus $\mathcal{D}$. i denotes the line number of the considered statement.

Rules (1), (2), and (3) of model $\mathcal{D}$ from Figure 1 originate from Hoare's calculus. Rule (4) is introduced because all information about variable values that are not changed by a certain statement should be propagated to the other statements. Of course there is the underlying assumption behind that the fault does not correspond to any defined variable. For example, if we assume the assignment statement `x = not(y);` to be faulty, then all information about the value of a variable `z` is passed by. Only information about the value of variable `x` is blocked. Note that Rule (4) also captures the case where s is a conditional statement. In this case all variables that are defined in any sub-block are not allowed to occur in predicate P.

In order to complete the modeling part, we have to provide a rule for loop statements. The Hoare calculus requires a loop invariant. Computing such an invariant is not always possible and for our purpose not necessary. Since we only consider one specific test case we know how often a loop statement has been executed simply by executing the program. Hence, we replace a loop statement by nested conditional statements and use the model for conditional statements instead. The statement `while` p `do` s_1 `end while` is represented by `if` p `then` s_1 `if` p `then` ... `else` ϵ `end if else` ϵ `end if` where ϵ denotes the empty statement block. A necessary condition for this transformation is that the program halts on the given input. Hence, faults related to infinite loops cannot be handled within our framework.

The second part of the framework comprises the formal definition of debugging and some consequences. Debugging herein is restricted to fault localization. As mentioned in the introductory example we define the result of debugging, i.e., the diagnoses, by checking consistency of derived or given predicates over a background theory. The background theory captures all knowledge about the domain of the variables, e.g., boolean algebra or arithmetic. In the following we assume that the required background theory is part of the program's model $\mathcal{D}$.

Definition 2 (Diagnosis). *Given a program Π, the corresponding diagnosis problem $(\mathcal{D}, \mathcal{T})$, and a set of indices $\mathcal{I}$ where each $x \in \mathcal{I}$ corresponds to a certain statement of Π. A set $\Delta \subset \mathcal{I}$ is a diagnosis iff*

$$\mathcal{D} \cup \mathcal{T} \cup \{Ab(x) \mid x \in \Delta\} \cup \{\neg Ab(x) \mid x \in \mathcal{I} \backslash \Delta\}$$

is consistent.

Since in most cases we are interested in diagnoses that are as small as necessary we introduce the concept of minimal diagnoses, i.e., diagnosis Δ is a minimal diagnosis if no proper subset $\Delta' \subset \Delta$ is a diagnosis. As already mentioned by Reiter [9] there only exists a diagnosis if $\mathcal{D} \cup \mathcal{T}$ itself is consistent. Considering our model $\mathcal{D}$ from Figure 1 some other properties stated in [9] unfortunately do not hold. For example, it is not possible to remove $\{Ab(x) \mid x \in \Delta\}$ and check consistency because $\mathcal{D}$ specifies the statement's behavior when assuming the statement to be faulty. Fortunately the following property still holds.

Property 1. Every superset of a diagnosis is itself a diagnosis.

Assume a diagnosis Δ and a set $\Delta' \supset \Delta$. For all elements in $\Delta' \setminus \Delta$ suppose the assumption about their state changes from $\neg Ab$ to Ab. Because of the definition of rule (4) (Fig. 1) this change has an influence on the derived predicates. However, only less predicates may be derived. Therefore, it is still not possible to cause an inconsistency.

Because of Property 1 it is enough to compute all minimal diagnoses. However, computing all minimal diagnoses is still NP-complete because we have to check all subsets of $\mathcal{I}$. A better way of computing all minimal diagnosis up to a given size which works in practice is the following. Instead of computing diagnoses directly, we compute conflicts. These conflicts can be used directly to compute all diagnosis up to a given size.

Definition 3 (Conflict). *Given a program Π, a corresponding diagnosis problem $(\mathcal{D}, \mathcal{T})$, and a set of indices $\mathcal{I}$ where each $x \in \mathcal{I}$ corresponds to a certain statement of Π. A set $\Xi \subset \mathcal{I}$ is a conflict iff*

$$\mathcal{D} \cup \mathcal{T} \cup \{\neg Ab(x) \mid x \in \Xi\} \cup \{Ab(x) \mid x \in \mathcal{I} \backslash \Xi\}$$

is inconsistent.

The relationship between conflicts and diagnoses can be explained as follows. Assume we have two conflicts $\Xi_1 = \{1, 3\}$ and $\Xi_2 = \{2, 3\}$. This means that assuming statements 1,3 respectively 2,3 to be correct (and the others to be faulty) leads to an inconsistency. Formally, we can write $\ldots \cup \{\neg Ab(1), \neg Ab(3)\} \cup \ldots \vdash \bot$ respectively $\ldots \cup \{\neg Ab(2), \neg Ab(3)\} \cup \ldots \vdash \bot$. In order to remove both conflicts we have to set the truth value of some Ab predicates to true which makes the literals $\neg Ab$ to become false and the contradiction $\bot$ can no longer be derived. The algorithm for computing diagnosis from conflicts which is basically a hitting-set computation is described in [9, 4]. This algorithm is also applicable in case of programs with multiple faults. Note that the computation of small conflicts can be improved by keeping track of the derivations. In most cases not all assumptions are involved in deriving an inconsistency.

3 Model Checking and Counterexamples

In order to compare our formal debugging model with an heuristic approach we briefly introduce the basic concepts of model checking and counterexamples [2]. The heuristic approach used for comparison stems from Groce and Visser [5].

Model checking of programs is based on a labeled transition system (LTS). An LTS is a 4-tuple $\langle S, S_0, Act, T \rangle$, where S is a finite non-empty set of states, $S_0 \subset S$ is the set of initial states, Act is the set of actions, and $T \subset S \times Act \times S$ is the transition relation. The program states in S are composed of a control location C and a data valuation D, and the partial functions $c : S \to C$ and $d : S \to D$ are defined. Furthermore the set of states S contains a distinguished set of error states $\Pi = \{\pi_1, \ldots, \pi_n\}$ representing assertion violations, uncaught exceptions, etc. Hence, the set of states is written as $S = (C \times D) \cup \Pi$. The transition relations $(s, \alpha, s') \in T$ are written as $s \xrightarrow{\alpha} s'$.

A finite transition sequence of length k starting in state $s_0 \in S_0$ is denoted as $t = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha} s_k$, where $0 < k < \infty$. A finite transition sequence $t = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha} s_k$ is a prefix of the finite transition sequence $t' = s'_0 \xrightarrow{\alpha'_1} s'_1 \xrightarrow{\alpha'_2} \cdots \xrightarrow{\alpha'\,,} s'_{k'}$ if $0 < k < k'$ and $\forall i \leq k.(i \geq 0 \Rightarrow s_i = s'_i) \wedge (i > 0 \Rightarrow \alpha_i = \alpha'_i)$. Similarly a control suffix of transition sequence t' is a finite transition sequence $t = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha} s_k$ if $0 < k < k'$ and $\forall i \leq k.(i \geq 0 \Rightarrow c(s_{k-i}) = c(s'_{k'-i})) \wedge (i > 0 \Rightarrow \alpha_{k-i} = \alpha'_{k-i})$. Based on an LTS and finite transition sequences the term counterexample is defined formally:

Definition 4. *A counterexample is a finite transition sequence t in the LTS $\langle S, S_0, Act, T\rangle$, where $s_0 \in S_0$, $s_k \in \Pi$, and $t = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha} s_k$.*

With respect to an initial counterexample t a so-called set of *negatives* is defined. It contains all counterexamples for the given LTS that result in the same error state from the same control location and is denoted by $neg(t)$ (or *neg* for short). The set is formally described by:

Definition 5. *A negative (w.r.t. a particular counterexample t) is a finite transition sequence from $s'_0 \in S_0$, $t' = s'_0 \xrightarrow{\alpha'_1} s'_1 \xrightarrow{\alpha'_2} \cdots \xrightarrow{\alpha'\,,} s'_{k'}$, where $0 < k' < \infty$, such that (1) $c(s_{k-1}) = c(s'_{k'-1}) \wedge \alpha_k = \alpha'_{k'}$, and (2) $s_k = s'_{k'}$.*

Similarly the set of *positives* is defined. It consists of executions that contain the statement immediately before the error but do not reach the error state and is denoted by $pos(t)$ (or *pos* for short).

Definition 6. *A positive (w.r.t. a particular counterexample t) is a finite transition sequence from $s'_0 \in S_0$, $t' = s'_0 \xrightarrow{\alpha'_1} s'_1 \xrightarrow{\alpha'_2} \cdots \xrightarrow{\alpha'\,,} s'_{k'}$, where $0 < k < \infty$, such that (1) $c(s_{k-1}) = c(s'_{k'-1}) \wedge \alpha_k = \alpha'_{k'}$, (2) $s'_{k'} \notin \Pi$, and (3) $\forall t'' \in neg(t) \cdot t'$ is not a prefix of t''.*

In general the sets $neg(t)$ and $pos(t)$ are potentially infinite. In order to keep the analysis tractable only subsets are generated. This requires that requirement 3 has to be relaxed: A positive must not be a prefix of any *found* negative. The algorithm for generating negatives and positives uses a model checker to explore backwards from a given counterexample. The search may be limited in case the state space is quite large or infinite. For a detailed definition and description of this algorithm refer to [5].

4 Case Study

The case study for the comparison of our formal approach and the heuristic approach is based on a small example program.

4.1 Example Program

The example program (Figure 2) used for a detailed explanation of the two approaches is an adaptation from Henzinger et al. [6]. A similar version of the

```
1  public static int LOCK = 0;
2  int got_lock = 0;
3  int count = 0;
4  do {
5      if (*) {
6          lock ();
7          got_lock++;
8      }
9      if (got_lock != 0) {
10         unlock ();
11     }
12     got_lock--; // faulty statement
13     count ++;
14 } while (*);
15 public static void lock () {
16     assert (LOCK == 0);
17     LOCK = 1;
18 }
19
20 public static void unlock () {
21     assert (LOCK == 1);
22     LOCK = 0;
23 }
```

Fig. 2. Example program

program is also used by Groce and Visser [5]. The program uses methods `lock()` and `unlock()` in order to acquire and release a lock. The lock is represented by variable `LOCK` which is set to 0 or 1 according to the state of the lock. The two locking methods assert that the lock is not held already if it is going to be acquired and that it is held if it is going to be released. The program keeps track of the state of the lock with the additional variable `got_lock`. A `*` in a condition of an if-statement or while-loop denotes a non-deterministic choice. A model checker that searches for violated assertions has to consider both possibilities.

The program in Figure 2 contains a fault with respect to its specification (i.e., the assertions). The decrement of variable `got_lock` in line 12 should be within the scope of the if-statement in line 9. This fault allows to acquire the lock twice and it is possible to release the lock although it is not held.

Compared to the version of the program used in [5] a third variable named `count` is added. It was introduced to force a larger state space that has to be explored by the model checker. The reason for this is that we want to evaluate the explaining counterexamples approach depending on the search depth. If variable `count` would be removed entirely from the program the state space explored by the model checker would be limited. Thus the number of positives and negatives would be limited as well.

4.2 Explaining Counterexamples

The work of Groce and Visser [5] is based on comparing negative and positive program traces. The analyses are intended to give an explanation on "what went wrong" in a given negative trace, i.e., a counterexample. Obviously this approach requires the existence of positive traces according to Definition 6. If no positives are found, the approach will not yield useful results.

The first analysis deals with sets of statements that occur in the negative and positive traces. Table 1 provides a definition of the sets that are used for the analysis. The interesting sets that are also given to the user are the *only*

Table 1. Transition analysis set definitions

Transition analysis set	Definitions
$trans(neg)$	$\langle c, \alpha \rangle \vert \exists t \in neg \cdot t$ contains $\langle c, \alpha \rangle$
$trans(pos)$	$\langle c, \alpha \rangle \vert \exists t \in pos \cdot t$ contains $\langle c, \alpha \rangle$
$all(neg)$	$\langle c, \alpha \rangle \vert \forall t \in neg \cdot t$ contains $\langle c, \alpha \rangle$
$all(pos)$	$\langle c, \alpha \rangle \vert \forall t \in pos \cdot t$ contains $\langle c, \alpha \rangle$
$only(neg)$	$trans(neg) \setminus trans(pos)$
$only(pos)$	$trans(pos) \setminus trans(neg)$
$cause(neg)$	$all(neg) \cap only(neg)$
$cause(pos)$	$all(pos) \cap only(pos)$

and *cause* sets. For instance the set $only(neg)$ denotes those statements that are unique to the negative traces, whereas the set $cause(neg)$ contains those statements that are necessary for a trace to be negative. Note that the statements contained in any of the sets strongly depend on the negatives and positives found by the explanation algorithm. The size of the sets varies between almost all statements of the entire program and the empty set.

The second analysis deals with the transformation of a positive trace into a negative trace. Therefore this analysis focuses on the sequence of statements instead of sets of statements. A transformation from a positive trace t into a negative trace t' is performed by replacing a block of statements occurring in t with a block of statements occurring in t'. Therefore it is necessary that traces t and t' consist of the same prefix and the same control suffix. A transformation from $t = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha} s_k$ into $t' = s'_0 \xrightarrow{\alpha'_1} s'_1 \xrightarrow{\alpha'_2} \cdots \xrightarrow{\alpha'} s'_{k'}$ exists if

1. $\exists p \cdot p$ is a finite transition sequence which is a prefix of both t and t', and
2. $\exists u \cdot u$ is a finite transition sequence which is a control suffix of both the largest prefix of t and the largest prefix of t'.

Note that in requirement 2 the largest prefix of transition sequences t and t' are taken, since the two traces do not share the same final control location, that is the negative trace t' ends in an error state whereas the positive trace t does not. The minimal transformation from t to t' is defined as a 3-tuple $\langle k_t, t_p, t_n \rangle$, where k_t is the length of the prefix in t (t'), and t_p (t_n) is the block of statements in t (t') that has to be replaced with t_n (t_p) in order to get a negative (positive) trace. The statement blocks t_n and t_p are again evaluated with the transition analysis defined in Table 1 in order to extract causal information for the fault. Note that for the second analysis *neg* and *pos* in Table 1 are replaced with t_n and t_p, respectively. Program traces are written as a sequence of statements, represented by their line number. Conditional statements containing a non-deterministic choice also contain the value that was chosen for the particular trace.

Our analysis is based on the counterexample with index 1 shown in Table 2. The traces found by the model checker with a limited search depth of 17 transitions include a negative trace (1) which is the initial counterexample itself, a positive trace (2), a second negative (3), a trace that ends in an assertion but is

Table 2. Traces found by model checker (search depth 17)

Index	Trace
1	$[1, 2, 3, 5F, 9, 12, 13, 14T, 5F, 9, 10, A]$
2	$[1, 2, 3, 5F, 9, 12, 13, 14T, 5T, 6, 7, 9, 12, 13, 14T, 5F, 9, 10, 12]$
3	$[1, 2, 3, 5F, 9, 12, 13, 14T, 5T, 6, 7, 9, 12, 13, 14T, 5F, 9, 10, 12, 13, 14T, 5F, 9, 10, A]$
4	$[1, 2, 3, 5F, 9, 12, 13, 14T, 5T, 6, 7, 9, 12, 13, 14T, 5T, 6, A]$
5	$[1, 2, 3, 5T, 6, 7, 9, 10, 12]$

not a valid negative (4), and a second positive trace (5). The first positive trace (2) has to be removed because it is a prefix of the second negative (3). Hence, the sets of negative and positive traces are $neg = \{1, 3\}$ and $pos = \{5\}$.

Based on this information the transition analysis sets are computed. Table 3 depicts the resulting sets. These are the same results as described in [5] except that the increment of variable `count` occurs in some sets. This is not a problem since it is easily removed form the explanation by performing some dependency analysis such as slicing [10]. Groce and Visser now conclude that the set $cause = \{5F, 12, 13, 14T\}$ notes the key points of the observed misbehavior: if the system chooses not to lock ($5F$), the decrement of `got_lock` (12) together with the loop being reiterated ($14T$) leads to an erroneous state. Thus it is possible to try to unlock even though the lock has not been acquired.

This indeed is an interesting result as it clearly describes the cause of the observed misbehavior. Unfortunately this information depends on the positive and negative traces found by the model checker. If the search depth is increased or decreased, the results look quite different. For example a search depth of 18 causes the model checker to find an additional positive trace and the analysis sets change as can be seen in Table 4. The analysis now has identified the if-statement in line 5 as the cause for the negatives. Unfortunately the real fault in the program is not present any more. The explanation that choosing `false` for the condition (i.e., not acquiring the lock) causes the error provides useful information to the user, but he still has to look at all statements that influence the locking/unlocking mechanism in order to locate the fault.

Even worse is the result of the analysis if the search depth is increased to 23. In this case the model checker identifies three negatives and four positives

Table 3. Transition analysis for search depth 17

Analysis set	Elements
$trans(neg)$	$\{1, 2, 3, 4, 5F, 5T, 6, 7, 9, 10, 12, 13, 14T\}$
$trans(pos)$	$\{1, 2, 3, 4, 5T, 6, 7, 9, 10\}$
$all(neg)$	$\{1, 2, 3, 4, 5F, 9, 10, 12, 13, 14T\}$
$all(pos)$	$\{1, 2, 3, 4, 5T, 6, 7, 9, 10\}$
$only(neg)$	$\{5F, 12, 13, 14T\}$
$only(pos)$	$\{\}$
$cause(neg)$	$\{5F, 12, 13, 14T\}$
$cause(pos)$	$\{\}$

Table 4. Transition analysis for search depth 18

Transition analysis set	Elements
$only(neg)$	$\{5F\}$
$only(pos)$	$\{\}$
$cause(neg)$	$\{5F\}$
$cause(pos)$	$\{\}$

Table 5. Transition analysis for search depth 6

Transition analysis set	Elements
$only(neg)$	$\{5F, 12, 13, 14T\}$
$only(pos)$	$\{5T, 6, 7\}$
$cause(neg)$	$\{5F, 12, 13, 14T\}$
$cause(pos)$	$\{5T, 6, 7\}$

Table 6. Transformation analysis results

S. depth	Analysis set	Elements
6	$only(t_n)$	$\{5F, 9, 12, 13, 14T\}$
	$only(t_p)$	$\{5T, 6, 7\}$
	$cause(t_n)$	$\{5F, 9, 12, 13, 14T\}$
	$cause(t_p)$	$\{5T, 6, 7\}$
17	$only(t_n)$	$\{5F, 9, 12, 13, 14T\}$
	$only(t_p)$	$\{5T, 6, 7\}$
	$cause(t_n)$	$\{5F, 9, 12, 13, 14T\}$
	$cause(t_p)$	$\{5T, 6, 7\}$

S. depth	Analysis set	Elements
18	$only(t_n)$	$\{5F\}$
	$only(t_p)$	$\{5T, 6, 710\}$
	$cause(t_n)$	$\{5F\}$
	$cause(t_p)$	$\{5T, 6, 7\}$
23	$only(t_n)$	$\{5F, 9, 12, 13, 14T\}$
	$only(t_p)$	$\{5T, 6, 7\}$
	$cause(t_n)$	$\{5F\}$
	$cause(t_p)$	$\{5T, 6, 7\}$

whereof two are prefixes of some negative. The analysis reveals that it is not possible to identify any statement as a cause for the negatives or positives. Further increasing the search depth leads back to one of the above mentioned cases, that is no positives are found, the cause of negatives contains some statements, and the cause of negatives is empty. Furthermore if the search depth is less then 17 it is possible to identify a cause for positives that is given in Table 5. In this case the cause for positives is identified as taking the lock and incrementing variable `got_lock`. This set adds almost no new information since it only confirms that not taking the lock causes an error.

In addition to the transition analysis sets we also performed the transformation analysis. Table 6 collects the results of the analysis. The results are comparable to those of transition analysis. Table 6 also shows that even if transition analysis reports no causal sets it is still possible to get this information from transformation analysis (e.g., search depth 23). Although, the results are not improved by transformation analysis. The main proposition gathered from transition and transformation analysis is the fact that not taking the lock in line 5 causes an error.

4.3 Applying Model-Based Debugging

Computing possible fault locations for the given example program with MBD requires test cases. Contrary to model checking it is not possible to specify values non-deterministically, instead values have to be specified according to a specific program execution. For the given example program this means that

Table 7. Diagnoses using different counterexamples

(a) Counterexample 1

Diagnoses
$[\text{LOCK} = 0]_1$
$[\text{got\_lock} = 0]_2$
$[\textbf{if}\ (*)]_5$
$[\textbf{if}\ (\text{got\_lock} != 0)]_9$
$[\text{unlock}\ ()]_{10}$
$[\text{got\_lock}--]_{12}$
$[\textbf{while}\ (*)]_{14}$

(b) Counterexamples 1 and 4

Diagnoses
$[\textbf{if}\ (*)]_5$
$[\textbf{if}\ (\text{got\_lock} != 0)]_9$
$[\text{got\_lock}--]_{12}$
$[\textbf{while}\ (*)]_{14}$

(c) Counterexamples 1, 3, and 4

Diagnoses
$[\textbf{if}\ (*)]_5$
$[\text{got\_lock}--]_{12}$
$[\textbf{while}\ (*)]_{14}$

the non-deterministic choices (∗) in lines 5 and 14 must be replaced with some expressions that enforce a predefined execution trace.

In order to compare the results of explaining counterexamples with the results of MBD the counterexamples found in Section 4.2 are translated into test cases. Since the example program uses neither input nor output variables, the test cases only include values for the conditions of the if-statement in line 5 and the while-statement in line 14. The assertion statements in lines 16 and 21 are also part of the model. Obviously, the resulting value of an assertion's expression is expected to be true. Therefore every test case will include an observation for assertions.

Table 7(a) depicts the set of diagnoses that is retrieved by applying MBD to the example program with the test case derived from counterexample 1. Note that all diagnoses are single fault diagnoses. This means that every statement represented by a diagnosis has a direct influence on the observed misbehavior (i.e., the violated assertion in line 21). The set of diagnoses is rather big compared to the overall size of the example program. This particular result could also be reproduced using a dynamic slicing algorithm [10]. But we show that this result can be improved using multiple test cases.

The results of MBD can be improved by providing additional test cases because of their potential to introduce new conflicts which make diagnoses more concise. The model checker found two additional counterexamples, as is shown in Table 2. Trace 4 is most promising to improve the results since it is very different from the first one. Contrary to trace 1 it violates the assertion in method `lock`. Thus traces 1 and 4 are selected for MBD. Performing diagnosis with these two traces yields a smaller set of diagnoses (Table 7(b)). The resulting diagnoses now must be able to explain the fault in *both* traces. Thus, diagnoses that may only explain the fault in one of the two traces are removed. Note that Table 7(b) only lists the single fault diagnoses, although there exist multiple fault diagnoses as well. But since multiple fault diagnoses are less likely to describe the real error they are omitted.

The set of diagnoses can be reduced slightly if the third counterexample, trace 3, is also used for MBD. In this case the diagnosis containing the if-statement in line 9 is removed. Adding further counterexamples to the diagnostic process

does not improve the results. The final set of diagnoses is shown in Table 7(c). This is the optimal result that we expect from MBD, since all three diagnoses that are left have a plausible explanation. It is possible to give a simple repair for each diagnosis that removes the fault with respect to the given specification. For example replacing the condition of the if-statement in line 5 with the constant `true` or the condition of the while-statement in line 14 with the constant `false` will prevent the violation of the assertions. Hence, it is not possible to get rid of these two diagnoses without any further specification. But most important the real fault (i.e., line 12) is among the diagnoses as well.

Considering the formal debugging framework based on resolution calculus the optimal result in terms of a minimal set of program statements has been found. It is not possible to improve the above result in terms of improving the diagnostic process or the used model. But still the result can be enhanced before it is presented to the user. For example it is very likely that conditional statements are part of the set of diagnoses. This stems from the way they are modeled for diagnosis. A conditional statement is responsible for selecting one of the possible branch values for each variable that is part of the conditional statement block. Therefore a conditional statement can rarely be removed from the diagnoses. Hence, a way to further improve MBD results is the application of some rating mechanism. Every type of diagnosis is assigned some probability value. The user would then be able to check the most likely diagnoses first. Obviously this is a heuristic approach that requires a well-founded analysis on the likelihood of faults in software systems.

The time consumed by the two approaches is quite similar, if we account searching for the positive and negative traces to the fault explanation approach as well. We conducted our evaluation on a Pentium 4 with 2.66 GHz using Java PathFinder (JPF) and a prototypical implementation of the value-based diagnosis model. Computing diagnoses for the example program in Figure 2 takes about 0.24 seconds on average and 2.4 seconds with overhead for parsing and model generation. The time for computing the initial counterexample with JPF takes 1.6 seconds on average. The time to search for additional negative and positive traces and computing the analysis sets is about 3.0 seconds. For larger programs, for instance a search tree program with approx. 150 lines of code using several insertion, deletion and search operations, diagnosis time ranges from 1 up to 10 seconds. The running time depends on the type and the location of a fault. With JPF the first counterexample was detected after 2.5 seconds on average. The analysis then lasted 3.5 seconds on average. Note that for the search tree example we provided a set of test cases for the model checker. Thus it was not necessary to search in a large state space to locate a counterexample.

5 Related Research and Conclusion

Software debugging and the application of automated tools to this task is an active research area. Detecting bugs using model checking [2] is becoming an accepted technique, providing witnesses of bugs called counterexamples and cor-

responding test cases. Several solutions to the challenge of localizing bugs based on these witnesses were proposed. Especially the model checking community applies heuristic approaches for localization. Common to all these approaches is their lack of a clear semantics for the reported fault locations.

In addition to the explanation approach discussed in this paper, Ball et al. [1] for instance use a model checker to identify the cause of an error. Their approach searches for counterexamples and compares them to a correct trace of the program. The main idea is that the differences between correct traces and erroneous traces identify the root cause of a fault. Those parts of the counterexample that are not part of the correct trace are replaced with `halt`-instructions. This prevents the model checker in subsequent runs to find the same counterexamples again. The additional counterexamples are used to refine the fault locations.

A similar work to fault explanation is the work of Zeller et al. [7, 12], *Delta debugging*. The motivation of their work is to answer the question why a program does not work anymore after changing it. The aim of delta debugging is to isolate a cause-effect chain that represents the variables and their values that caused the failure. Based on a failing and a correct program execution, delta debugging minimizes the difference between the two runs in terms of isolating the relevant input and states. Analogous to explaining counterexamples this approach relies on the existence of successful program runs that are similar to the failing ones.

Model-based diagnosis is related to program-slicing [10] as is stated by Wotawa [11]. In case of a dependency model for diagnosis it is shown that slices are equivalent to conflict sets. In our debugging calculus the relation to slicing appears in terms of Hoare's rules, since a predicate is only transformed by a statement if it has an effect on the predicate. This is similar to the slicing approach using weakest preconditions proposed by Comuzzi and Hart [3].

We presented a formalization of model-based debugging based on a resolution calculus. Contrary to other debugging approaches we provide a formal model for describing program behavior and do not rely on any heuristics. The formality of the model and the underlying diagnosis theory allows to assign a precise semantics to the diagnoses (i.e., fault locations) reported by the approach. This is an essential difference to heuristic approaches where no clear semantics for a reported cause is given. The quality of heuristic approaches diverges depending on many parameters. The same fault produces different causes depending on the position of the fault within the program, the program traces used for explanation and the time spent for explanation (i.e., the search depth). Contrary, our approach does not make any structural assumptions except the implicit fault model. Adding more information (e.g., more counterexamples) leads to more precise diagnoses, whereas the results of heuristic approaches may become worse.

The presented approach contains some limitations. The debugging calculus used to model program behavior induces a specific fault model. That is, statements that are assumed to be faulty do not affect predicates that do not contain any target variable of that statement. This fault model restricts the set of faults that are precisely localizable with our approach. In case of a faulty assignment statement with the wrong target variable our approach derives wrong predi-

cates. Nevertheless, debugging experiments based on this calculus have shown that such structural faults are localized by our approach, but only in diagnoses with higher cardinality. This stems from the fact that a wrong target variable in general provokes a wrong value in two variables, the actual target variable and the intended target variable.

Faults that result in non-terminating programs can not be handled at all. This mainly stems from modeling of loop statements by unrolling them into nested conditionals. The number of loop iterations must be known to generate the behavioral model of a program.

References

1. T. Ball, M. Naik, and S. K. Rajamani. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, volume 38, 1 of *ACM SIGPLAN Notices*, pages 97–105, Jan. 2003.
2. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
3. J. J. Comuzzi and J. M. Hart. Program Slicing using Weakest Preconditions. *Lecture Notes in Computer Science*, 1051:557–575, 1996.
4. R. Greiner, B. A. Smith, and R. W. Wilkerson. A Correction to the Algorithm in Reiter's Theory of Diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.
5. A. Groce and W. Visser. What Went Wrong: Explaining Counterexamples. In *SPIN Workshop on Model Checking of Software*, pages 121–135, May 2003.
6. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, volume 37, pages 58–70, Jan. 2002.
7. R. Hildebrandt and A. Zeller. Simplifying Failure-Inducing Input. *SIGSOFT Software Engineering Notes*, 25(5):135–145, 2000.
8. C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, 1969.
9. R. Reiter. A Theory of Diagnosis from First Principles. *Artificial Intelligence*, 32(1):57–95, 1987.
10. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.
11. F. Wotawa. On the Relationship between Model-Based Debugging and Program Slicing. *Artificial Intelligence*, 135(1–2):124–143, 2002.
12. A. Zeller. Isolating Cause-Effect Chains from Computer Programs. In *ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE-10)*, pages 1–10, Charleston, South Carolina, Nov. 2002.

A Technique to Represent and Generate Components in MDA/PIM for Automation*

Hyun Gi Min and Soo Dong Kim

Department of Computer Science,
Soongsil University,
511 Sangdo-Dong, Dongjak-Ku, Seoul, 156-743, Korea
hgmin@otlab.ssu.ac.kr, sdkim@ssu.ac.kr

Abstract. Component-Based Development (CBD) is an effective approach to develop software effectively and economically through reuse of software components. Model Driven Architecture (MDA) is a new software development paradigm where software is generated by a series of model transformations. By combing essential features of CBD and MDA, both benefits of software reusability and development automation can be achieved in a single framework. In this paper, we propose a UML profile for specifying component-based design in MDA framework. The profile consists of UML extensions, notations, and related instructions to specify elements of CBD in MDA constructs. Once components are specified with our profile at the level of PIM, they can be automatically transformed into PSM and eventually source code implementation.

1 Motivation

MDA is a new software development paradigm where a model plays a key role in automatic software development [1]. It provides a systematic framework to understand, design, operate, and evolve all aspects of enterprise system, using engineering methods and tools. The framework is based on modeling different aspects and levels of abstraction of such systems, exploiting interrelationships between these models.

A very common technique for achieving platform independence is to target a system model for a technology-neutral virtual machine. A model in PIM is reusable over different platforms. Hence, we regard PIM as neither executable unit nor implemented unit. PIM enables models to be traced and improves maintainability through modifying model and regeneration into PSM.

CBD is another promising approach to develop software system effectively and economically through reuse of software components. Especially, domain-common components provide a common set of features and functions in a domain, so that application members can utilize the components by customizing the behavior with minimum effect.

* This work was supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD). (KRF-2004-005-D00172).

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 293 – 307, 2006.

Therefore, if MDA is combined with CBD approach, we can acquire a highly effective development environment where the commonality and variability(C&V) in a domain are modeled and developed as MDA compatible components, and software development can be greatly automated. Moreover, since C&V is reflected in designing PIM and code-level components are not limited to only one platform, the reusability of components is greatly increased.

In this paper, we suggest techniques to combine the advantage of CBD and MDA. We first define a component-based PIM (CB-PIM) and proposed a UML profile for specifying component design in MDA/PIM. The profile consists of UML extensions, notations, and related instructions to specify elements of CBD in MDA constructs. If components are designed by using the proposed method, the design can be automatically transformed into source code implementation, yielding benefits of reusability and automation.

2 Foundation

2.1 Model Driven Architecture (MDA)

MDA is an approach to using models in software development. The essence of MDA is making a distinction between Platform Independent Models (PIMs) and Platform Specific Models (PSMs). To develop an application using MDA, it is necessary to first build a PIM of the application, then transform this using a standardized mapping into a PSM, and finally map the latter into the application code by automation.

The three primary goals of MDA are portability, interoperability and reusability through architectural separation of concerns [1]. Some of the motivations of the MDA approach are to reduce the time of adoption of new platforms and middleware, primacy of conceptual design, and interoperability. The MDA approach makes it possible to save the conceptual design and the MDA helps to avoid duplication of effort and other needless waste [2][3].

2.2 UML Profile

A UML profile defines standard UML extensions that combine and/or refine existing UML constructs to create a dialect that can be used to describe artifacts in a design or implementation model. The UML profile defines a set of UML extensions that capture the structure and semantics. It defines several standard extension mechanisms, including stereotypes, constraints, tagged values and icons [4]. When one defines a profile, it is common MDA practice to also define mappings that specify how to transform models conforming to the profile into artifacts appropriate to the kinds of systems. If a model is not specified by a particular UML profile, the model can not be transformed automatically by MDA mechanism.

The OMG has adopted a MOF metamodel of Java and EJB to complement the UML profile for EJB [5], a UML profile for modeling enterprise application integration [6] and a UML profile for CORBA [7] as well. However, these profiles only support implementation levels and do not present component of a PIM level.

2.3 Fontoura's UML-F

UML-F is an UML extension that supports working with object-oriented frameworks and allows the explicit representation of framework variation points [8]. A framework, UML-F assumes, is a collection of several fully or partially implemented components with largely predefined cooperation patterns between them.

This framework implements the software architecture for a family of applications with similar characteristics, which are derived by specialization through application-specific code. UML-F suggests constraint *{appl-class}, {variable}, {extensible}, {static}, {dynamic}, {incomplete}, {for all new methods}* and *{optional}*.

However, elements are not explicitly identified in this model and no precise definition for the elements is suggested. Only the overall meaning of a framework that UML-F reference is explained.

2.4 Exertier's Component Design PIM

Exertier suggests a 'Component Design PIM' that represents a platform-independent solution expressed in terms of software components [9]. The modeling of the distributed components PIM includes four major activities. *Partition the system:* The architecture of a software subsystem identifies a set of architectural elements, here components, which collaborate to achieve the system's functional and non-functional requirements. The objective of this activity is to specify this decomposition. *Perform the component boundary design*: As defined by UML2.0, a component is a modular, deployable and replaceable (pluggable) part of a system. It encapsulates its internal part and exposes a set of interfaces. *Perform the component internal design*: When the boundary of a component has been defined, its internal design can be performed. *Perform the components logical deployment*: Components collaborate to reach functional and non-functional requirements of the subsystem.

This research only suggests four activities for designing component as a PIM. However, it does not deal with how to specify each activity and variability of component for PIM.

2.5 Kim's Variation Types

Kim's work establishes a theoretical foundation on variability in component based development by defining five types of variability and three kinds of variability scope [10]. In this, various variability-related terms are defined such as *Variation Point (VP)*, *Variant*, and *Variability*. Also, five types of variability in CBD are identified; variability on *Attribute*, *Logic*, *Workflow*, *Interface* and *Persistence*. Attribute is defined as an abstract storage to store values, and it is realized as constants, variables, or data structures.

Attribute variability denotes occurrences of *variation points* on attributes. Logic describes an algorithm or a procedural flow of a relatively fine-grained function. *Logic variability* denotes occurrences of *variation points* on the algorithm or logical procedure. *Workflow variability* denotes occurrences of *variation points* on the sequence of method invocations. Persistency is maintained by storing attribute values of a component in a permanent storage so that the state of the component can alive over

system sessions. *Persistency variability* denotes occurrences of *variation points* on the physical schema or representation of the persistent attributes on a secondary storage.

3 Elements of Component Design

In this section, we define elements of a component design and each element is elaborated in details.

A *component* is defined as a set of related *classes*, and it provides a relatively coarse-grained functionality as Fig. 1. All the *classes* in a component are related in some way; association, inheritance, aggregation, composition, and dependency. *Operations* available through the *interface* of components are generally larger-grained than methods in a class. The behavior of these operations is modeled as a *workflow*, which is a sequence of method invocations among the objects/classes in a component.

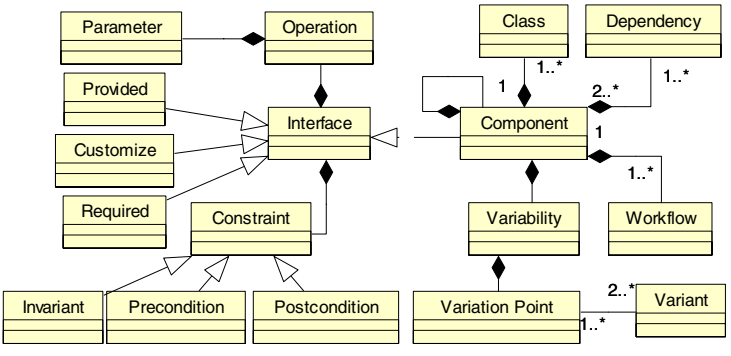


Fig. 1. The metamodel of component

An *interface* has one or more operations, and each operation is given a signature that consists of the operation name, input parameters and a return type. Semantics of each operation should be described to define the behavior and constraints of the operation. It is described by a pre-condition, a post-condition, an invariant, side effects, and constraints. A post-condition describes the state of an object that should be met after an operation finishes execution. A side effect of an operation is any additional changes in the state of related objects besides the main object.

Variabilities are characteristics that may vary from application to application. In general, all variabilities can be described in terms of alternatives. Variability is defined as variation points and variants. Modeling and realizing variability is one of the unique features of CBD. Variability is characterized by a number of variations within the common requirement. It consists of variation points and all their valid variants for variable requirement that is determined to have a minor and detailed difference among some family members by relevant stakeholders.

A variation point identifies one or more locations in a software asset at which the variation will occur [11]. Griss defines variation point as an explicitly designated location within a component at which a variability mechanism may be used to create a

customized component [12]. A Variation Point is a place in software where the minor difference occurs for variable requirement. A Variant is a value or instance that can validly fill in a variation point, i.e. a variant resolves a variation point.

A software quality model is a specification of software quality attributes and their relationship. ISO 9126 is a representative quality model for generic software [13]. A quality attribute is a non-functional characteristic of a component or a system, such as integrability, usability, efficiency, modifiability, reliability, security, transaction, flexibility or availability. Also, deployment of component affects performance, reliability, security, availability, capacity and bandwidth. The component is an executable unit. Therefore, we need not only functionality of component but also extra functional information that supports components deploying and operating.

4 Component Development Process Using UML Profiles

In this section, we propose a component development process using UML profile for specifying components to improve the applicability of PIM of component level as Fig. 2. Analysis process extracts functional and non-functional requirements. The analyzed requirements are represented using UML 2.0 by object oriented design process. This process yields PIMs based on objects.

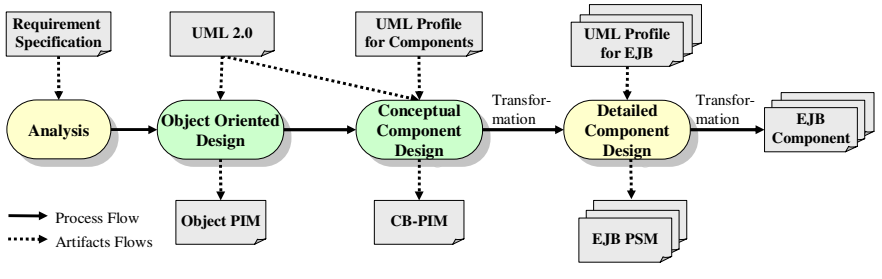


Fig. 2. Component Development Process using UML Profile for Components

In the conceptual component design, the PIMs of object level transform into component-based PIM (CB-PIM) that presents general component information. The general component information that is units, interfaces, variability, and environments of components does not depend on component platforms such as EJB, CORBA, etc. This process identifies the general component information. None of these can be represented by UML 2.0 [19]. Therefore, we need to UML profile for specifying components to present these. The UML profile will be introduced later. Object PIM transforms into CB-PIM that is not dependent on component platform such as EJB and CORBA.

In the detailed component design, the CB-PIM can be automatically transformed into each PSM using the UML profile for component platforms such as UML profile for EJB. Finally, the generated PSMs are transformed into each component source. Therefore, traditional MDA process reuses the object level of PIM. Our MDA process

reuses the component level of CB-PIM. Once components are specified with our profile at the level of PIM, they can be automatically transformed into PSM and eventually source code implementation.

5 UML Profile for Specifying Components

In this section, we suggest a UML profile for specifying components. Our UML profile to represent CB-PIM is based on the UML 2.0. Elements from UML 2.0 and EDOC are used in our profile; the elements for CB-PIM that are not supported by UML 2.0 [14] are extended from MOF. Our UML profile is MOF. Therefore, the CB-PIM that is specified by our profile can be presented by common MDA tools.

5.1 UML Profile for Specifying Component Units

In CBD, a component is the fundamental unit of packaging related objects [12], hence we need to specify the related objects in a component in PIM. A port is a connection point between a classifier and its environment. Connections from the outside world are made to ports according to provided and required [15]. Workflow in a component can be designed by sequence and communication diagrams according to UML 2.0. The UML profile for specifying component units is presented as Table 1.

Table 1. The Elements of UML Profile for Component Units Design

Element	Presentation	Applies to	Remarks
Component	«component»	component	Use UML 2.0
System Component	«SysComponent»	component	
Business Component	«BizComponent»	component	
Transient Class	«Transient»	class	
Persistence Class	«Persistence»	class	Default
Primary Key	«UniqueId»	attribute	
Synchronous Message	«Sync»	method	Default
Asynchronous Message	«Async»	method	
Message Call	«use», «call», etc.	dependency	Use UML 2.0
Relationships	association, inheritance, composition, aggregation, dependency	relationship	Use UML 2.0
Constraints	{ }, pre:, post:, inv:	class, method, relationship, etc.	Use OCL
Algorithm	Use Text	method	Use OCL, ASL

Components are in general classified into *system* components and *business* components [16]. A system component interacts with client programs and manages client transactions by coordinating message flows among participating components and/or objects which mostly manipulate data. System components provide a system service that is the external representation of the system, providing access to the

services of the system. This service acts as a façade and a mediator for the business service [17]. A business component consists of persistent objects which handle persistent business data. Hence, business components execute upon the requests from system components. To denote two types of components in PIM, we use stereotypes; «SysComponent» and «BizComponent».

Persistency objects that should be stored in database or file systems are represented by a stereotype «Persistence». If some objects such as value objects [17] for transforming data are not persistency, a stereotype «Transient» is used. Asynchronous messages use a stereotype «Async» that are described at methods in class, sequence, and communication diagrams. Constraints and algorithms can be expressed by Object Constraints Language (OCL), and Action Semantic Language (ASL).

As Fig. 3 shows, the *LoanMgr* component is denoted as a system component with «SysComponent» stereotype, and composed of one class. The *LoanAccount* component is denoted as a business component with «BizComponent» stereotype, and its two member classes are shown.

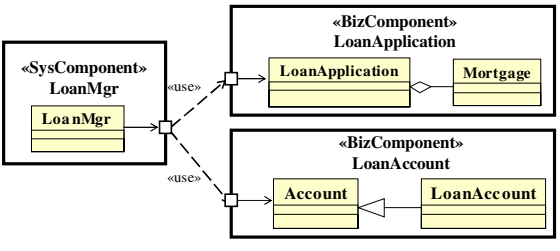


Fig. 3. Example of Expressing Component Units

5.2 UML Profile for Specifying Interfaces

A component provides its component-level interface, i.e. the protocol for accessing the service of the component. In CBD, an interface is clearly separated from component implementation to increase the maintainability and replaceability [12]. Hence, we need to specify some interfaces as well as component units in CB-PIM as Table 2.

Table 2. The Elements of UML Profile for Interface Design

Element	Presentation	Applies to	Remarks
Interface	«Interface»	Interface	Use UML 2.0
Provided Interface	«ProvidedInterface»	Interface	Use UML 2.0
Customize Interface	«CustomizeInterface»	Interface	
Required Interface	«RequiredInterface»,	Interface	Use UML 2.0
Signature	operationName(param:Type): ReturnType	Operation	Use UML 2.0
Constraints	{ }, pre:, post:, inv:	Class, Method, Relationship	OCL
Algorithm	Use Text	Method	OCL, ASL

In CBD, three types of interface can be modeled; *provided*, *customize* and *required* interfaces. The *provided* interface specifies the services provided by a component and it is invoked by other components or client programs at runtime. The stereotype «ProvidedInterface» is used to denote this interface, and the name of the *provided* interface is defined by using 'Ip' prefix name.

Components often provide mechanisms to tailor the behavior of the components through an interface designed especially for this purpose. A *customize* interface consists of methods that are used to assign a *variant* to a *variation point* [18]. To specify *customize* interface, we use a stereotype «CustomizeInterface» and 'Ic' prefix on the name of the *customize* interface.

The *required* interface specifies external services invoked by the current component, i.e. a specification of external services required by the current component [18]. By specifying the *required* interface for a component, we can precisely define the services invoked by the current component. This information can be later used in integrating related components into an application or a component framework. The *required* interface can be specified with a stereotype «RequiredInterface». An interface consists of operation signatures and their semantics. The semantics can be expressed in terms of pre- and post-conditions and invariants using OCL.

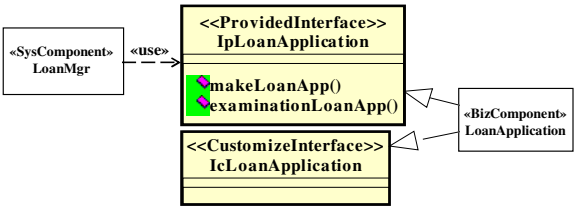


Fig. 4. Example of Expressing Interfaces

Fig. 4 shows an example of expressing interfaces and their realized components in CB-PIM, where a *LoanApplication* component is realized by IpLoanAppication interface and IcLoanApplication interface. The IpLoanApplication interface is a required interface of LoanMgr component. The LoanMgr component requests services of the LoanApplication component. The required interface of the LoanMgr is the IpLoanApplication.

5.3 UML Profile for Specifying Variation

The commonality and variability is made explicit through variation points and variants in the components and other reusable component elements [12]. The goal is to create a set of reusable components that expresses commonality and variability appropriate to the family of applications.

The variability can increase the reusability of component. However, the UML does not support notations of variability. Therefore, variability is designed by non-standard stereotypes, tagged values, or note elements [20]. If the variability is presented by standard notation, MDA tools identify variation points by the

variability. The variation points of PIM or PSM can be filled by other design artifacts and automation tools.

We define types of variation that are attribute variability, logic, workflow, persistency and interface variability as in [10]. To express variation points of a component in CB-PIM, we propose «VP-Attr», «VP-Logic», «VP-WF», «VP-Persistency» and «VP-Interface» stereotypes as in Table 3.

Table 3. UML Profile for Variation Design

Element	Presentation	Applies to	Remarks
Variation Point (VP)	«VP»	Attribute, Method	
Attribute VP	«VP-Attr»	Attribute, Use case	
Logic VP	«VP-Logic»	Method	
Workflow VP	«VP-WF»	Method	
Interface VP	«VP-Interface»	Operation	
Persistency VP	«VP-Persistency»	Operation, Method	
Variant	«Variant»	Class, Operation, Method	
Variation Scope	{vScope = value}	Variation Point	Close, Open
VP ID	{vpID = value}	Variation Point, Variant	
Variant ID	{varID = value}	Variant	
Constraints	{ }, pre:, post:, inv:	Class, Method, Relationship	OCL
Algorithm	Use Text	Class, Method	OCL, ASL

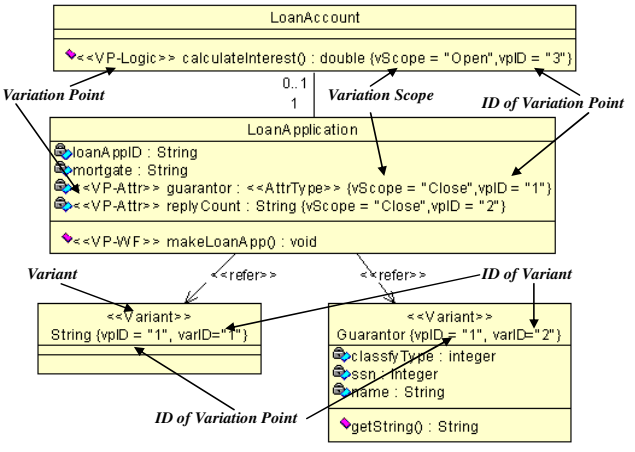


Fig. 5. Example of Expressing Variation

We present two kinds of scope of variation points. *Open* scope of variation point has any number of variants which are already known and additional variants which

are currently unknown but can possibly be found later at customization or deployment time. In constraint, *close* scope of variation point has two or more variants which are already known [10].

Fig. 5. shows an example of expressing variability in CB-PIM. The logic of *calculateIntereste()*can be changed by each family member. The class 'LoanApplication' has two variation points which are guarantor and replyCount. Two variants of the attribute guarantor are a type String and a class Guarantor. The attribute guarantor has variation that has two variants; String and object Guarantor. If the variant string is set as {varID="1"}, the attribute has string data type to store guarantor's ID. If the object Guarantor is set, the data type of the attribute becomes Guarantor. In the implement process, the variation will be implemented by the value of varID later.

5.4 UML Profile for Specifying Extra-Function

A component is an executable unit. We need not only functionality of components but also extra functionality of components that supports components deploying and operating. The extra functional properties extensions are motivated more by the desire to ensure that interface specifications are sufficiently complete to ensure correct integration than by the desire to extend the scope of information hiding to additional properties. Both ends are served by these extensions [21].

To specify extra functional information in CB-PIM more practically, we classify properties into four types; deploy property, runtime property, transaction property and security property as Table 4 . A deploy property captures information for deploy on server. A runtime property specifies runtime environment for component instances. A transaction property defines method of transaction. A security property manipulates strategy about usage of component.

For example, the stereotype «DeployProperty » specifies information for deploy environment. An attribute deployedName as align is called and managed by component middleware server. When the component is running in a server, the mechanism of the component server may use the align name. The components are packaged automatically by the artifactName attribute.

Table 4. UML Profile for Extra Functional Design

Element	Presentation	Applies to	Remarks
Deployment Property	«DeploymentProperty»	Component	Stereo type
	deployedName, artifactName		Tagged Value
Runtime Property	«RuntimeProperty»	Component	Stereo type
	virtualClientsPerInstance, instancePerComponent, instanceTimeToLive componentTimeToLive, instanceInactivityTimeout		Tagged Value
Transaction Property	«TXProperty»	Component, Interface, Class, Method	Stereo type
	useTX, TXAttrType, TXIsolation, TXTimeOut		Tagged Value
Security Property	«SecurityProperty»	Component, Interface, Class, Method	Stereo type
	userRoleName		Tagged Value

Our UML profile represents an activation policy that describes how a client gains access to the component, whether it has exclusive access to the component, and certain lifetime restrictions on the component. The stereotype «RuntimeProperty» specifies information for runtime environment and lifetime restrictions. The activation constraints that can currently be specified in a runtime property are: limits on the number of clients per-instance and per-component, restrictions on the number of instances per-component, limits on the time an instance or a component may exist, including an inactivity timeout, the name by which clients may activate the component and activation operations which allow parameterized activation of the component.

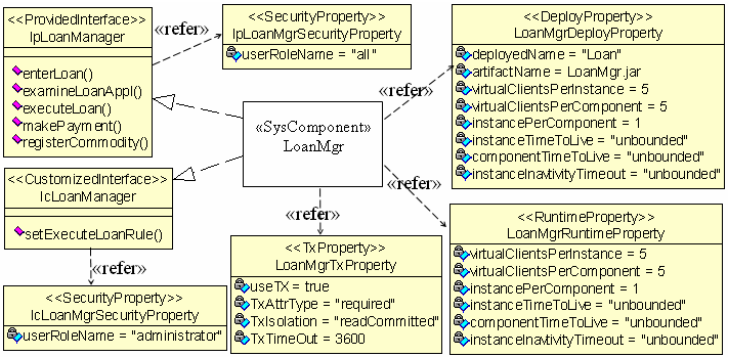


Fig. 6. PIM showing Deploy Property of LoanMgr Component

The stereotype «TxProperty» specifies strategy about transaction of component. If useTX attribute is false, other transaction attributes are ignored by PSM or Code level. The attribute TxAttrType has a TxAttrTypeKind enumeration type. The TxAttrTypeKind enumeration type consists of required, requiredsNew, supports, mandatory, notSupported and never value. The attribute of TxIsolationType has a TxIsolationTypeKind enumeration type. The TxIsolationTypeKind enumeration type consists of readUncommitted, readCommitted, repeatableRead and serializable. The TxTimeOut is a timeout period for transaction operation. If the transaction access time is over TxTimeOut, the transaction should be rolled back.

The stereotype «SecurityProperty» contains strategy about security of component. The attribute of userRoleName is the permitted role name of a component's caller. The role of the component is assigned by this userRoleName attribute. The customize interface may used by an administrator. In this case the userRoleName attribute of the «SecurityProperty» is an 'administrator'. The provided interface may used by all customers. This userRoleName is an 'all'. This attribute may apply to <security-role-ref>, <security-role> and <method-permission> in the deployment descriptor at PSM level for EJB. Fig. 6 shows an example of expressing extra functional property of a LoanMgr system component in CB-PIM.

6 Assessment

The Fountoura's UML-F [8] is based on object framework. Elements are not explicitly identified in this model and no precise definition for the elements is suggested. UML-F reference only explained the overall meaning of a framework. Exertier's research [9] only suggests four activities for designing components with a PIM. This research does not deal with how to specify each activity and the variability of components with a PIM. The UML 2.0 and UML profile for EDOC [22] does not fully present the profiles for specifying general component.

Table 5. Comparing the suggested UML Profile with others (✔: Supported)

Technique / Factor	Comp. Spec.	UML 2.0	EDOC Profile	Our Profile	Remarks
Component Units	✔	✔	✔	✔	«SysComponent», etc.
Provided Interface	✔	✔	✔	✔	«ProvidedInterface»
Required Interface	✔	✔		✔	«RequiredInterface»
Customize Interface	✔			✔	«CustomizeInterface»
Variation Point	✔			✔	«VP-Attr», etc.
Variant	✔			✔	«Variant», etc.
Non Functional Design	✔			✔	«TXProperty», etc.
Workflows	✔	✔	✔	✔	Sequence Diagram, etc.
Reusing Model Level		✔	✔	✔	PIM Level

Our profile covers variability and extra functional designs as well as the four designs of Exeriter's component design PIM such as partition of the system, component boundary design, component internal design, and components logical deployment as in Table 5. Therefore, once components are specified with the suggested UML profile for specifying components at the level of CB-PIM, they can be automatically generated each source code implementation as shown in Fig.7. A CB-PIM can be reused into diverse platforms.

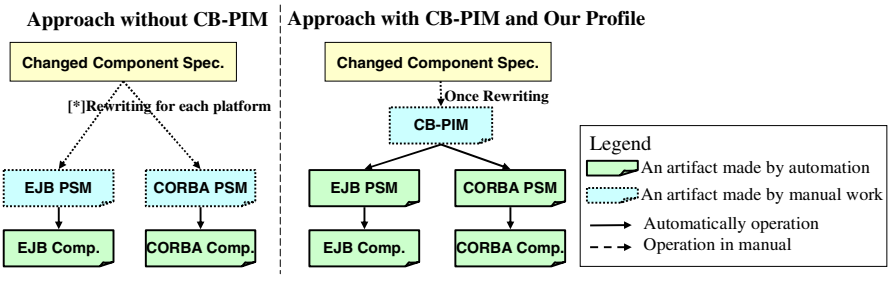


Fig. 7. An advantage of CB-PIM and UML Profile for Specifying Components

If the mechanism for implementing components which is shown in Fig.7 is supported with a tool, various components such as EJB and CORBA can be effectively implemented by using the seamless method and tools. To make our approach more practical and useful, we are developing a prototype development tool based on Eclipse as Fig. 8. The prototype will support all our UML profile and the mechanism.

Eclipse can plug modules such as our component designer and code generator prototypes as in Fig. 8. The component designer based on Graphical Editor Framework (GEF) [23] stores the PIM models to extended UML2 file for our profile. UML2 [24] is an EMF-based implementation of the UML™ 2.0 metamodel for the Eclipse platform. The code generator transfers the UML2 file to codes by using XMI schema. Eclipse basically includes a code editor. Therefore, components can be specified with our UML profile for specifying components at the level of CB-PIM. CB-PIM can be automatically transformed into each PSM and eventually each source code implementation by use the tool.

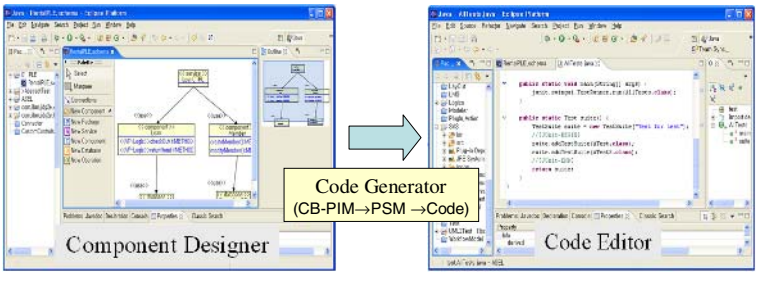


Fig. 8. Component Designer based on Eclipse

7 Conclusion Remarks

CBD is to develop software system effectively and economically through reuse of software components. Effective components should be designed using interfaces, component units, variability, and non-functional factors for components. As a basic reuse unit, components often come in black-box form, only exposing well-defined interface while hiding internal details.

MDA is a n approach to using models in software development. The essence of MDA is making a distinction between PIM and PSM. To develop an application using MDA, it is necessary to first build a PIM of the application, then transform this using a standardized mapping into a PSM, and finally map the latter into the application code by automation.

If a component's middleware is changed but requirement is not modified, the related components should be redesigned and re-implemented because components platforms are diverse. If component specifications are designed at MDA/PIM, we can automatically create the components that are satisfied by the component platform of an application. Therefore, we need the UML profile for specifying components to make machine-understandable design for MDA tools.

In this paper, we proposed a UML profile for specifying component-based design and component development process in MDA framework. Our UML profile consists of UML extensions, notations, and related instructions to specify elements of CBD in MDA constructs. It can be presented by general UML and MDA design tools. Once components are specified in the proposed profile at the level of PIM, they can be automatically transformed into PSM and eventually source code implementation by MDA tools. By using the UML profile for specifying components, we believe that the productivity, reusability, applicability, and maintainability of components can be greatly increased by automation.

References

[1] OMG, "MDA Guide Version 1.0.1," *omg/2003-06-01*, June 2003.

[2] Flater., D., "Impact of Model-Driven Architecture," *In Proceedings of the 35th Hawaii International Conference on System Sciences*, January 2002.

[3] Frankel, D. and Parodi, *The MDA Journal, Model Driven Architecture Straigth from the Masters*, Meghan-Kiffer Press, 2004.

[4] Frankel, D., *Model Driven Architecture™:Applying MDA™ to Enterprise Computing*, Wiley, 2003.

[5] Java Community Process, UML Profile For EJB_Draft, 2001.

[6] OMG, "UML™ Profile and Interchange Models for Enterprise Application Integration(EAI) Specification," 2002.

[7] OMG, "UML Profile for CORBA Specification V1.0, OMG," Nov. 2000.

[8] Fontoura, M., Pree, W., and Rumpe, B., "UML-F: A Modeling Language for Object-Oriented Frameworks," *Lecture Notes in Computer Science Vol. 1850*, 2000.

[9] Exertier, D., Lnaglois, B., and Roux, X., "PIM Definition and Description," *Proceedings of* 1st *European Workshop, Model-Driven Architecture with Emphasis on Industrial Applications(MDA-IA 2004),* 2004.

[10] Kim, S., Her, J., and Chang, S., "A theoretical foundation of variability in component-based development," *Information and Software Technology, Vol. 47*, *p.663-673*, July, 2005.

[11] Muthig, D. and Atkinson, C., "Model-Driven Product Line Architectures," *Lecture Notes in Computer Science Vol. 2379*, pp.110-129, 2002.

[12] Heineman, G. and Councill, W., *Component-Based Software Engineering*, Addison Wesley, 2001.

[13] Kim, S. and Park, J., "C-QM: A Practical Quality Model for Evaluating COTS Components", *IASTED International Conference on Software Engineering (SE'2003),* 2003.

[14] Choi, S., Chang, S., and Kim, S., "A Systematic Methodology for Developing Component Frameworks," *In Proceedings of 7th Fundamental Approaches to Software Engineering (FASE'04) Conference, Lecture Notes in Computer Science Vol. 2984*, 2004.

[15] Rumbaugh, J., Jacobson, I., and Booch, G., *The Unified Modeling Language Reference Manual, Second Edition*, Addison-Wesley, 2004.

[16] Cheesman, J. and Daniels, J., *UML Components*, Addison-Wesley, 2001.

[17] Roman, E., *Mastering Enterprise JavaBeans™ and the Java™2 Platform*, Enterprise Edition, WILEY, 1999.

[18] Kim, S., Min, H., and Rhew, S., "Variability Design and Customization Mechanisms for COTS Components," *Lecture Notes in Computer Science Vol. 3480,* pp. 57-66, 2005.

[19] OMG, *Unified Modeling Language: Superstructure version 2.0,* ptc/03-08-02, 2003.
[20] Geyer, L. and Becker, M., "On the Influence of Variabilities on the Application-Engineering Process of a Product Family," " *Lecture Notes in Computer Science Vol. 2379*, 2002.
[21] Bachman, F. and Bass, L., "Volume II:Technical Concepts of Component-Based Software Engineering," *CMU/SEI-2000-TR-008*, May 2000.
[22] OMG, *UML Profile for EDOC V1.0*, 2004.
[23] Eclipse Project, *Graphical Editor Framework (GEF),* at URL: http://www.eclipse.org/gef/
[24] Eclipse Project, *UML2*, at URL: http://www.eclipse.org/uml2/

Argus: Online Statistical Bug Detection*

Long Fei[1], Kyungwoo Lee[1], Fei Li[2], and Samuel P. Midkiff[1]

[1] School of Electrical and Computer Engineering
[2] School of Industrial Engineering,
Purdue University,
West Lafayette, IN 47907, USA
{lfei, kwlee, li64, smidkiff}@purdue.edu

Abstract. Statistical debugging is a powerful technique for identifying bugs that do not violate programming rules or program invariants. Previously known statistical debugging techniques are offline bug isolation (or localization) techniques. In these techniques, the program dumps data during its execution, which is used by offline statistical analysis to discover differences in passing and failing executions. The differences identify potential bug sites. Offline techniques suffer from three limitations: (i) a large number of executions are needed to provide data, (ii) each execution must be labelled as passing or failing, and (iii) they are post-mortem techniques and therefore cannot raise an alert at runtime when a bug symptom occurs. In this paper, we present an *online statistical bug detection* tool called Argus. Argus constructs statistics at runtime using a *sliding window* over the program execution, is capable of detecting bugs in a single execution and can raise an alert at runtime when bug symptoms occur. Moreover, it eliminates the requirement for labelling all executions as passing or failing. We present experimental results using the Siemens bug benchmark showing that Argus is effective in detecting 102 out of 130 bugs. We introduce optimization techniques that greatly improve Argus' detection power and control the false alarm rate when a small number of executions are available. Argus generates more precise bug reports than the best known bug localization techniques.

1 Introduction

Program correctness and reliability are two of the greatest problems facing the developers, deployers and users of software. The financial implications are tremendous – a recent NIST report estimates that $59.6 billion dollars a year, or 0.6% of the GDP, are lost every year because of software errors [17]. Single failures of software can disable businesses like Charles Schwab and eBay for hours or days, costing millions of dollars in revenues. Moreover, purely malicious and politically motivated sociopaths exploit software errors to disrupt civil society. Simultaneous with the increasing risks of bug-ridden code has been the rise in

* This work was supported by the National Science Foundation by grants 0325603-CCR and 0313033-CCR, and by the Indiana Utility Regulatory Commission. The opinions expressed are those of the authors, and not the National Science Foundation.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 308–323, 2006.

size and complexity of software, which makes software debugging an increasingly challenging task. In order to relieve the programmers from laborious debugging, automated debugging techniques have been extensively studied. Among these, *bug detection* techniques focus on determining the existence/nonexistence of a bug; *bug isolation/localization* techniques focus on determining the cause of observed bug symptoms. In both cases, an effective bug report that distinguishes the potential bug site(s) (or their vicinities) will greatly expedite the debugging process. While the effectiveness of a bug isolation tool is typically measured by its precision in locating bug sites, the effectiveness of a bug detection tool is measured by three factors: the detection rate, the false alarm rate, and the precision with which it locates bug sites. The detection rate reveals the power of the tool in determining a bug has occurred; the false alarm rate measures the likelihood of raising an alarm when there is no bug; and the precision in locating bug sites measures the quality of a bug report generated by the tool.

Statistical debugging techniques have recently caught the attention of the debugging research community. These techniques isolate software bugs by comparing the data collected from a large number of passing and failing executions using statistical methods (e.g. logistic regression [13], conditional probability [14], and hypothesis testing [15]). Because these techniques identify differences in runtime information that is correlated with program behaviors, they have the potential to isolate software bugs that do not violate programming rules or program invariants – the bugs that are most difficult to isolate using traditional debugging techniques.

However, existing statistical debugging techniques share some drawbacks. First, they require a large number of passing and failing executions to facilitate offline statistical analysis. While the passing executions may be obtained from in-house regression testing before software release, a large number of failing executions are usually unavailable. This requirement has three consequences in reality. First, the debugging process relies on the Internet to collect data from a large number of executions, which restricts the applicability of these techniques in devices with limited connectivity and increases the complexity of the software. Second, collecting execution data from customers raises the question of sensitive information leaking, which restricts the applicability of these techniques in sensitive environments. Third, the software vendor cannot respond timely to a product failure until it accumulates enough failures. This contradicts the ethics of debugging in that the customers have to suffer many more failures (and the resulting damage) even though the software vendor is well aware of the existence of a bug after the first observed failure!

Second, existing statistical debugging techniques require labelling passing and failing executions. This requirement has two consequences. First, it is expensive to construct test cases and their corresponding expected results, and requires extensive domain knowledge. Second, it is infeasible to apply these techniques to applications where the correctness of an outcome can not be easily verified or the expected outcome is unknown *a priori*. Although a program crash is a typical symptom of failure, many failures are non-crashing.

Third, existing statistical debugging techniques are *postmortem* techniques, i.e. they are bug isolation/localization techniques, not bug detection techniques. They cannot raise an alert when bug symptoms occur. An alarm at the time of failure can give the user more time to respond to the failure and to reduce the damage caused by the failure. This is particularly important in mission-critical programs where an incorrect operation can initiate a chain of catastrophic consequences.

In this paper, we present an online statistical *bug detection* technique, called Argus. Argus detects software bugs at runtime using an *anomaly detection* approach. Argus overcomes the three drawbacks mentioned above by constructing runtime statistics on a *sliding window* over an execution rather than the whole execution, which makes it possible to accumulate multiple observations in a single execution. Argus runs in two modes: training and detection. In the training mode, Argus builds statistical models from one or multiple passing executions. In the detection mode, Argus can detect bugs in a single execution. Argus does not require labelling of passing and failing executions. Argus can sound an alarm at runtime when bug symptoms occur. Our experiments show that Argus is effective in detecting 102 out of 130 bugs in the Siemens bug benchmark. We also introduce optimization techniques to greatly improve Argus' bug detection power and achieve a low false alarm rate when a small number of detection executions are available. Argus generates a precise bug report that outperforms the best known bug localization techniques on the Siemens benchmark. Our experiments on the SPEC2000 benchmark show that Argus runtime overhead is comparable to that of many existing successful tools.

We make the following contributions:

- We propose the first online statistical bug detection technique that can detect bugs with a single execution. We present its design and implementation.
- We propose mathematically-rigorous formulas and their implementations to maximize the detection rate, to minimize the false alarm rate, and to minimize the number of executions required to meet a given detection rate and a given false alarm rate when multiple executions are available.
- We present the experimental results using the Siemens bug benchmark and SPEC2000 showing the effectiveness of our technique.

Supplemental discussions and code used in this paper can be found on the Argus web site (`https://engineering.purdue.edu/Apollo/research/argus`).

2 Related Work

Existing runtime bug detection techniques can be placed into two categories [25]: *programming-rule based* and *statistical-rule based*. Tools in the former category (e.g. `Purify` [10], `SafeC` [1], and runtime assertion) check runtime violations against programming language specifications, programming paradigms, or user-specified program-specific rules (e.g. dereferencing a `NULL` pointer or memory leak); tools in the latter category check runtime violations against program invariants (e.g. value invariants [7, 8], PC invariants [25]). Runtime statistical-rule

based checking differs from Argus in that the judgement in runtime statistical-rule based checking is based on whether a single sample violates an existing invariant, while the judgement of Argus is based on whether a runtime *statistic* computed from *an aggregation of samples* fits an existing statistical model.

Recently several statistical bug isolation/localization tools have been developed. Earlier work by Burnell et al. use a Bayesian belief network as a supplement to program slicing to find the root of a bug by analyzing a program dumpfile [2]. State of the art techniques collect runtime information from a large number of executions, and apply statistical techniques in offline data mining to discover the differences between passing and failing executions. Liblit et al. use logistic regression to select suspicious value predicates associated with a bug [13], and later use a technique based on probabilistic correlations of value predicates and program crash [14]. Liu et al. propose a technique based on hypothesis testing in [15].

Statistical methods have been used for other debugging problems. Dickinson et al. find program failures through clustering program execution profiles [6]. Podgurski et al. use logistic regression to select features and cluster failure reports on selected features [18]. Cluster results are shown to be useful in prioritizing software bugs.

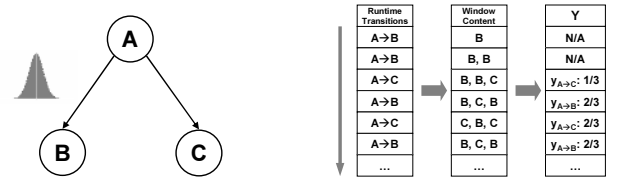
Sekar et al. detect anomalous program behavior by tracing system calls using an FSA-based approach [22]. Dallmeier et al. propose a postmortem defect localization technique for Java based on the differences in frequencies of certain class method calling sequences showing up in passing and failing runs [5]. A sliding window is used to divide the calling sequences into subsequences for efficiency in comparison.

3 Argus Statistical Model

Argus detects bugs at runtime using an *anomaly detection* approach. Argus collects samples of runtime statistics characterizing program's runtime behavior. In the training run(s), samples of these runtime statistics are used to build statistical models. In the detection run, samples of these runtime statistics are tested against the learned statistical models. If the samples significantly deviate from the learned statistical models, an alarm is raised.

3.1 Extended Finite State Automaton

We use an extended finite-state-automaton (ext-FSA) to characterize the program's runtime control-flow behavior. In our ext-FSA, each state is a *runtime event*, and each transition is a transition from one runtime event to the next runtime event. Runtime events are defined to be the set of {`program start`, `program exit`, `procedure entrance`, `procedure return`, `loop start`, `loop exit`, `compound statement entrance`, `compound statement exit`}. Each transition is augmented with the distribution of transition frequency (Figure 1(a)). We assume that anomalous transition patterns are indicators of buggy behaviors.



(a) Extended Finite-State-Automaton

(b) Argus Runtime Statistics Collection (window size = 3)

Fig. 1. Argus Statistical Model

3.2 Modelling State Transition Statistics

Using the states in Figure 1(a) for example, we define random variable $X_{A\to B}$ to be: $X_{A\to B} = 1$ if transition $A \to B$ takes place; and $X_{A\to B} = 0$ if transition $A \to C$ takes place, where C is a state other than B. The random variable $X_{A\to B}$ has an unknown distribution $\Theta_{X,A\to B}$. Let $\{X_{A\to B,i}\}$ be an *independent and identically distributed (i.i.d)* sample from the distribution. According to Central Limit Theorem [3], the following statistic

$$Y_{A\to B} = (\Sigma_{i=1}^{n} X_{A\to B,i})/n \tag{1}$$

conforms to a normal distribution $N(\mu_{Y_{\to}}, \sigma^2_{Y_{\to}})$ as $n \to \infty$, with mean $\mu_{Y_{\to}}$ and variance $\sigma^2_{Y_{\to}}$. Intuitively, $X_{A\to B}$'s distribution is dictated by the conditional probability: $Pr\{transitionA \to B|stateA\}$; and $Y_{A\to B}$ is the *frequency* of transition $A \to B$ being taken in the past n accesses to A, which is an estimator of the conditional probability: $Pr\{transitionA \to B|stateA\}$ based on past n accesses to A.

To determine if an observation $y_{A\to B}$ comes from an existing model $\Theta_{Y,A\to B}$ (in our case $N(\mu_{Y_{\to}}, \sigma^2_{Y_{\to}})$), we adopt a *hypothesis testing* [3] approach. Our *null hypothesis* H_0 is: $y_{A\to B}$ belongs to $N(\mu_{Y_{\to}}, \sigma^2_{Y_{\to}})$. Let the *probability density function (pdf)* of $Y_{A\to B}$ be $f_{Y_{\to}}(y|\theta_{Y_{\to}})$, where $\theta_{Y_{\to}}$ denotes the parameters of the normal distribution. The *likelihood function* of observation $y_{A\to B}$ coming from $N(\mu_{Y_{\to}}, \sigma^2_{Y_{\to}})$ is

$$L(\theta_{A\to B}|y_{A\to B}) = f_{Y_{\to}}(y_{A\to B}|\theta_{Y_{\to}}) \tag{2}$$

We reject the null hypothesis if the likelihood is smaller than a threshold, which indicates that $y_{A\to B}$ is significantly different from the existing model.

We make several assumptions to make this method practical for bug detection. First, we assume that $Y_{A\to B}$ computed from a window containing K consecutive $X_{A\to B}$'s (when $n = K$ in Equation 1) approximately conforms to a normal distribution, where K is a system parameter (window size). Second, we assume $Y_{A\to B}$'s we compute from different windows are *i.i.d.* samples from $Y_{A\to B}$'s underlying distribution. Our experiments show that these assumptions are reasonable approximations of the theoretical properties.

In order to perform runtime bug detection, Argus computes runtime statistics using Equation 1 on a *sliding window* of size K rather than the full execution

trace. For each state A, let S_A be the set of states that are possible targets of transitions from A. At runtime, if we observe transition $A \rightarrow B$, where $B \in S_A$, we compute $y_{A \rightarrow B}$ by applying Equation 1 on the last K transitions from A. Figure 1(b) shows how Argus computes y's at runtime. In this example, K is set to 3. A window size of 10 works well in our experiments.

We augment each transition $A \rightarrow B$ in our ext-FSA with a distribution of the corresponding statistic $Y_{A \rightarrow B}$. Mean and variance of the distribution are estimated from samples of the statistic $\{y_{A \rightarrow B,i}\}$ collected in training executions:

$$\mu_{Y_{A \rightarrow B}} = (\Sigma_{i=1}^{N} y_{A \rightarrow B,i})/N \tag{3}$$

and

$$\sigma^2_{Y_{A \rightarrow B}} = \frac{1}{N-1} \Sigma_{i=1}^{N} (y_{A \rightarrow B,i} - \mu_{Y_{A \rightarrow B}})^2 \tag{4}$$

where N is the number of samples. During a detection run, each $y_{A \rightarrow B}$ is tested against the null hypothesis. For simplicity, we test the normalized statistic:

$$Z_{A \rightarrow B} = (Y_{A \rightarrow B} - \mu_{Y_{A \rightarrow B}})/\sigma_{Y_{A \rightarrow B}} \tag{5}$$

against the null hypothesis "$H_0^{'}$: $z_{A \rightarrow B}$ belongs to $N(0,1)$[1]". The corresponding pdf of this normalized statistic is $f(z) = \frac{1}{\sqrt{2\pi}} e^{-z^2/2}$. For computational efficiency, we measure the *log likelihood* and ignore the constant coefficient $\frac{1}{\sqrt{2\pi}}$:

$$logL(\theta|z) = -z^2/2 \tag{6}$$

If the log likelihood falls below a threshold t, an alarm is sounded to indicate the possible existence of a bug.

For each state A, frequencies of transitions to different states in S_A are usually *inversely correlated*. For example, in Figure 1(a), large $Y_{A \rightarrow B}$ leads to small $Y_{A \rightarrow C}$, and vice versa. Therefore, Argus tests the log likelihood of $y_{A \rightarrow B}$ only when $y_{A \rightarrow B} >= \mu_{Y_{A \rightarrow B}}$, where $\mu_{Y_{A \rightarrow B}}$ is the mean of the learned distribution. The underlying strategy is that anomalously small $y_{A \rightarrow B}$ values will be reflected by an anomalously large $y_{A \rightarrow C}$ and thus Argus only needs to sound one alarm.

3.3 Alarm Threshold

Given a program P, Argus can be considered as a *binary function* of an execution e and the log likelihood threshold t. $Argus_P(e,t)$ yields 1 if Argus raises alarm on e, and 0 otherwise.

Given a set of failing executions $\{e_i\}_{i=1...m}$, and a log likelihood threshold t, we define the *detection rate* of Argus to be:

$$D(t) = (\Sigma_{i=1}^{m} Argus_P(e_i,t))/m \tag{7}$$

[1] For simplicity and computational efficiency, here we use a normal distribution to approximate Z's distribution(t-distribution).

Intuitively, $D(t)$ is the fraction of buggy executions that Argus raises an alarm. Accordingly the *false alarm rate* given a set of passing executions $\{e'_i\}_{i=1...n}$ and t is defined to be the fraction of normal executions that Argus raises alarms:

$$F(t) = (\Sigma_{i=1}^{n} Argus_P(e'_i, t))/n \tag{8}$$

Intuitively, the larger the threshold t is, the larger $D(t)$ and $F(t)$ are. Therefore, the detection power of Argus is largely constrained by the number of false alarms the user is willing to tolerate. Let f be the false alarm limit, we have $t = F^{-1}(f)$, and the corresponding detection rate is a function of f:

$$D^{'}(f) = D(F^{-1}(f)) \tag{9}$$

In Argus, t is a tunable system parameter. F can be found by measuring the false alarm rates with different t's using the passing executions from the test cases before software release. This allows the user to tune Argus based on the false alarm rate he/she is willing to tolerate.

3.4 Optimizing the Effectiveness over Multiple Executions

Although Argus is capable of detecting bugs using a single detection execution, Argus' power is *not* confined to a single execution. Argus can benefit from multiple detection executions to improve the overall detection rate and to reduce the overall false alarm rate. Now the question is: if multiple detection executions are available, how can a user *maximize the overall detection rate*, *minimize the overall false alarm rate*, and *minimize the number of detection runs needed*?

Let $\{f_i\}_{i=1...m}$ be the set of different false alarm rates (sorted in ascending order), with f_m being the upper bound on the false alarm rate the user is willing to tolerate. The corresponding detection rates are $\{d_i\}_{i=1...m}$. Let R denote the total number of detection executions, and r_i denote the number of detection executions under false alarm rate f_i, $\Sigma_{i=1}^{m} r_i = R$. We have the *overall miss rate*:

$$M(\{d_i\}, \{r_i\}) = \Pi_{i=1}^{m} (1 - d_i)^r \tag{10}$$

which is the joint probability of the bug going undetected in all the detection executions. Correspondingly, the *overall detection rate* is:

$$D(\{d_i\}, \{r_i\}) = 1 - M(\{d_i\}, \{r_i\}) = 1 - \Pi_{i=1}^{m} (1 - d_i)^r \tag{11}$$

Although actual $\{d_i\}$ are not known *a priori*, they can be estimated by running Argus against bugs found and fixed in the testing phase.

Problem 1: Given $\{f_i\}$, $\{d_i\}$, R, and overall false alarm rate limit f, how are values for $\{r_i\}$ chosen to maximize the overall detection rate?

Rather than solving this problem, we attack the dual problem instead: how do we minimize the overall miss rate. This problem can be described as the following *linear programming* problem:

$$\begin{aligned} \text{minimize: } & \Sigma_{i=1}^{m} r_i log(1 - d_i) \\ \text{subject to: } & \Sigma_{i=1}^{m} r_i = R, 0 \leq r_i \leq R \\ & \Sigma_{i=1}^{m} f_i \times r_i \leq f \times R \end{aligned} \tag{12}$$

Here we take the *log* of $M(\{d_i\},\{r_i\})$ to make it linear. The constraints are: the total number of executions under different false alarm limits is R, and the total number of false alarms under different false alarm limits must be smaller than the total number of false alarms allowed. The optimal detection rate is a function of $\{f_i\}$, $\{d_i\}$, f, and R under the constraints:

$$D_{opt}(\{f_i\},\{d_i\},f,R) = \underset{\{r\ \}}{argmax}(1 - exp(\Sigma_{i=1}^{m} r_i log(1-d_i))) \tag{13}$$

which uses the result from Equation 12.

Problem 2: Given $\{f_i\}$, $\{d_i\}$, R, and overall detection rate requirement d, how do we choose values for $\{r_i\}$ to minimize the overall false alarm rate?

The corresponding linear programming problem is:

$$\begin{aligned} \text{minimize: } & \Sigma_{i=1}^{m} f_i \times r_i \\ \text{subject to: } & \Sigma_{i=1}^{m} r_i = R, 0 \le r_i \le R \\ & \Sigma_{i=1}^{m} r_i log(1-d_i) \le log(1-d) \end{aligned} \tag{14}$$

Here we minimize the total number of false alarms under the constraints the total number of executions is R, and the overall miss rate must be smaller than $(1-d)$. The optimal false alarm rate is a function of $\{f_i\}$, $\{d_i\}$, R, and d under the constraints:

$$F_{opt}(\{f_i\},\{d_i\},R,d) = \underset{\{r\ \}}{argmin}(\frac{1}{R}\Sigma_{i=1}^{m} f_i \times r_i) \tag{15}$$

which uses the result from Equation 14.

Problem 3: Given $\{f_i\}$, $\{d_i\}$, the overall false alarm rate limit f and the overall detection rate requirement d, how many detection runs are sufficient?

The corresponding linear programming problem is:

$$\begin{aligned} \text{minimize: } & \Sigma_{i=1}^{m} r_i \\ \text{subject to: } & 0 \le r_i, i = 1 \ldots m \\ & \Sigma_{i=1}^{m} f_i \times r_i \le f \times \Sigma_{i=1}^{m} r_i \\ & \Sigma_{i=1}^{m} r_i log(1-d_i) \le log(1-d) \end{aligned} \tag{16}$$

where the constraints are: total number of false alarms must be smaller than the number of false alarms allowed, and the overall miss rate must be smaller than $(1-d)$. The optimal number of executions needed is a function of $\{f_i\}$, $\{d_i\}$, f, and d under the constraints:

$$R_{opt}(\{f_i\},\{d_i\},f,d) = \underset{\{r\ \}}{argmin}(\Sigma_{i=1}^{m} r_i) \tag{17}$$

which uses the result from Equation 16.

Equations 12, 14, and 16 are easily solvable using linear programming tools. Sample code written in `GAMS` (`http://www.gams.com`) can be found on the Argus web site.

4 Argus Design and Implementation

Argus is implemented as a runtime library for `C`. We use the `Cetus C` compiler [12] to instrument the program source with calls to the Argus runtime library. `Cetus` instruments the following program points: program start, program exit, procedure entrance, procedure return, loop entrance, loop exit, compound statement entrance, and compound statement exit. Argus monitors the transitions of these events, and computes the runtime statistic y for each transition using Equation 1. Argus intercepts crashing signals (`SIGSEGV`, `SIGABRT`, and `SIGTERM`), and monitors event transitions even at a program crash.

If Argus is running in training mode, it also estimates the distribution parameters for each transition. Because Equations 3 and 4 require storing every y (sample of the runtime statistic Y) computed using Equation 1 at runtime, they are not scalable. Instead, we estimate the distribution parameters *recursively* using the following equations [24]:

$$\mu_t = \frac{t-1}{t}\mu_{t-1} + \frac{1}{t}y_t \tag{18}$$

$$\sigma_t^2 = \frac{t-1}{t}\sigma_{t-1}^2 + \frac{1}{t-1}(y_t - \mu_t)^2 \tag{19}$$

The estimated distribution parameters are dumped into a transition distribution profile at the end of each training execution. The profile is then loaded at the initialization phase of a detection execution.

If Argus is running in detection mode, it performs likelihood test for each y computed at runtime using Equations 5 and 6. A bug report is generated at the end of each detection run. The bug report contains the top k most suspicious transitions (transitions with lowest log likelihood values), where k is a configurable parameter.

5 Experiments

In this section, we evaluate the effectiveness of Argus at detecting bugs, its runtime overhead, and the profile size. To test Argus' bug detection power, we apply Argus to the Siemens bug benchmark [11]. To measure the cost of applying Argus to real world applications, we use the SPEC2000 benchmark to measure Argus' runtime overhead and profile size.

The Siemens benchmark was originally used by Siemens Corporation Research to study test adequacy criteria [11]. The siemens benchmark contains seven small programs: `print_tokens, print_tokens2, replace, schedule, schedule2, tcas,` and `tot_info`. Each program has multiple versions. Each version is injected with one bug. Altogether there are 130 faulty versions, simulating a wide variety of realistic bugs. A number of previous works in the field of bug isolation have reported their results using this benchmark, including [4, 9, 15, 19, 20, 21].

5.1 Bug Detection Power

We measure the bug detection power of Argus on the Siemens benchmark. Our goal is to reveal the average detection rate and the effectiveness of Argus with regard to given false alarm limits. We consider Argus *effective* in detecting bugs in a faulty program if the alarm raised by Argus is more likely to be a true alarm rather than a false alarm. We define the *effective function* on a given faulty program P as (using ternary expression semantics):

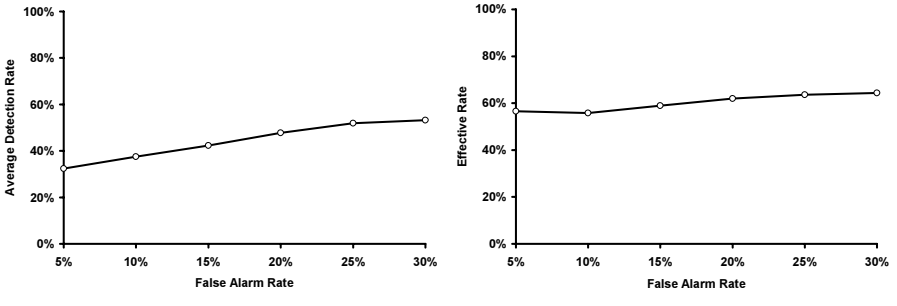
$$E_P(f) = (D'(f) > f \ ? \ 1 \ : \ 0) \tag{20}$$

where $D'()$ is defined in Equation 9. That is, given a false alarm rate f, if the corresponding detection rate is larger than f, we consider Argus effective. Given a set of faulty programs $\{P_i\}_{i=1...m}$, and a false alarm rate f, the *effective rate* is defined as:

$$ER_{\{P\}}(f) = \frac{1}{m} \Sigma_{i=1}^{m} E_P\,(f) \tag{21}$$

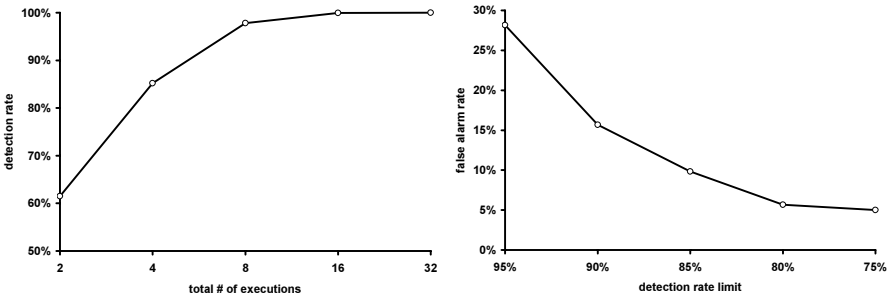
Intuitively, the effective rate measures the overall effectiveness of Argus on a set of faulty programs. Specifically, our goal is to establish Equation 9 and Equation 21. We use 10 as our sliding window size.

Each program in the Siemens benchmark has a bug-free version. We partition the test cases randomly with a 9 : 1 ratio as follows. We apply Argus in training mode on the bug-free version with 90% of the test cases to learn a statistical model for each transition. We then apply Argus in detection mode on the bug-free version with the remaining 10% of the test cases. The smallest log likelihood value observed in each execution is recorded and sorted in ascending order. Among the sorted log likelihood values, the one with x% of the values smaller than it is recorded as the log likelihood threshold corresponding to false alarm limit of x%. Intuitively, if we use this as the threshold, Argus will sound an alarm on x% of the normal executions. We discover the log likelihood thresholds corresponding to false alarm rates {5%, 10%, 15%, 20%, 25%, 30%} (assuming 30% is the upper limit of user tolerance) in this way. Each application has its own set of thresholds.



(a) Average Detection Rate on All Faulty Versions (b) Effective Rate on All Faulty Versions

Fig. 2. Argus Bug Detection Power



(a) Overall False Alarm Rate = 10% (b) Total Number of Executions = 4

Fig. 3. Argus Overall False Alarm Rate, Overall Detection Rate, and Total Number of Execution

We note that Argus does not rely on bug-free version to discover thresholds. Thresholds discovered in this way are equivalent to thresholds discovered on test cases where no bug sites are covered, which corresponds to the whole passing test suite before software release in real world. In our experiment, an alternative is discovering the thresholds for each faulty version using passing cases on that particular version. We consider it less proper because the passing cases in each faulty version may not be a general representative of normal cases (while the whole test suite is assumed to be general).

We then measure the detection rates under these thresholds by running each of the faulty versions of each application through *all* the test cases. For each faulty version, Argus (in training mode) learns a transition distribution profile from the passing executions. We then use the set of thresholds corresponding to that application discovered above, and measure on how many of the failing executions Argus (in detection mode) raises alarms under the false alarm limits {5%, 10%, 15%, 20%, 25%, 30%}. The detection rate under a false alarm limit is then computed using Equation 7. The average detection rate over the 130 faulty versions with regard to different false alarm levels are shown in Figure 2(a).

Effectiveness (Equation 20) is measured by comparing the detection rate on each faulty version with the corresponding false alarm limit. The effective rate (Equation 21) is then measured by averaging the results of Equation 20 on all the 130 faulty versions. The average effective rate is shown in Figure 2(b).

Argus sounds an alarm on 118 out of 130 faulty versions. On average, Argus has 32.89% chance to detect the bug in each faulty version using a single detection run when the false alarm tolerance is 5%. This average detection rate increases almost linearly to 52.27% when the false alarm tolerance is 25%. It is further increased to 53.54% when the false alarm tolerance is 30%. When the false alarm tolerance is {5%, 10%, 15%, 20%, 25%, 30%}, Argus is *effective* (as defined in Equation 20) in detecting bugs in {74, 73, 77, 81, 83, 84} of the faulty versions. Altogether, 102 out of 130 bugs can be *effectively* detected by Argus under at least one of the false alarm rate configurations.

Using the $\{f_i\}$ and average $\{d_i\}$ values discovered above, we can maximize the detection rate, minimize the false alarm rate, and assess how many executions are sufficient when multiple executions are available. Equations 13,15 and 17 can be written in short forms as $D_{opt}(f, R)$, $F_{opt}(R, d)$, and $R_{opt}(f, d)$, respectively. These three functions can be plotted with 3D charts. For simplicity and readability reasons, we project them onto 2D charts by fixing one variable at a reasonable value and then plot the relationship between the remaining two (Figure 3). Interested readers can find other values of interest using the code provided on Argus web site. Due to space constraints, we present only two of the projections. The remaining projections can be found on Argus web site. From Figure 3, we can see that when the false alarm tolerance is 10%, 8 executions are sufficient to guarantee a 98% detection rate; if we have only 4 runs, we can let the overall detection rate slip to 80% to bring the overall false alarm rate down to 6%. These results show that the detection power of Argus increases rapidly with a small number of executions, and the false alarm rate can be kept to a low level. This in turn implies that, in general, using Argus can greatly reduce the number of test cases needed, leading to a shorter debugging cycle.

5.2 Bug Report Quality

To measure the quality of the bug report generated by Argus, we adopt a paradigm which was proposed by Renieris et al. [20] and was later adopted by Cleve et al. in `CT` [4] and Liu et al. in `SOBER` [15]. This paradigm estimates the human effort to locate the bug based on the bug report. The measure was based on the size of the subgraph of the program dependence graph (PDG) the human must explore (starting from the statements pointed out by the bug report) vs. the size of the full PDG. Because Argus detects anomalous *transitions* rather than *statements*, we use a variant of the paradigm:

1. The FSA is a connected graph G, where each state is a vertex and each transition is an edge.
2. If $A \rightarrow B$ is a transition, and if the buggy statement is executed after event A (with no other events in between) yet before event B (with no other events in between), we consider $A \rightarrow B$ a *defect edge*. Note that one buggy statement may cover multiple defect edges depending on runtime control flow. We use V_{defect} to denote the set of vertices covered by defect edge(s).
3. Given a bug report, the set of vertices covered by the top k suspicious transitions reported by Argus is denoted V_{blamed}.
4. A programmer performs a breadth-first search from V_{blamed} until a vertex in V_{defect} is reached. The set of vertices covered by the breadth-first search is denoted $V_{examined}$.
5. The T-score is defined as:

$$T = |V_{examined}|/|V| \times 100\% \tag{22}$$

where $|V|$ is the size of graph G.

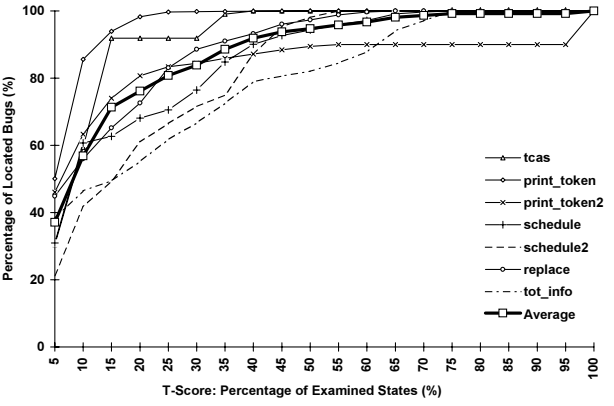


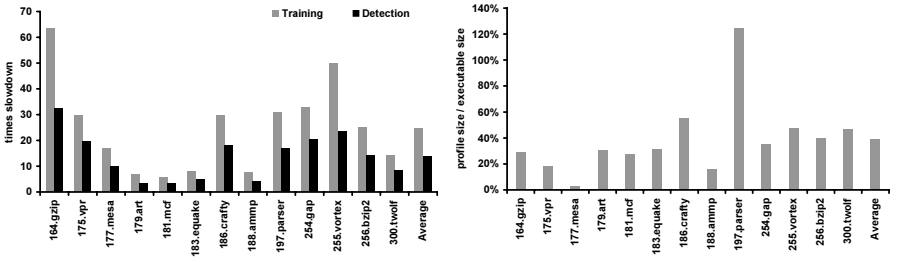
Fig. 4. Argus Bugs Located (cumulative) vs. Average T-score, k = 5

The T-score *estimates* the human effort to reach the buggy transition from the top k suspicious transitions reported by Argus. A high T-score corresponds to more human effort, and in the worst case a score of 100% indicates the whole program must be examined. We measure the T-score of Argus bug report for *each buggy execution* in each faulty version.

Although Argus' major design goal is runtime bug detection, the bug report it generates is precise enough that it can be used for efficient bug localization as well. Figure 4 shows the number of bugs that can be localized vs. the average T-score (percentage of states examined in ext-FSA, as defined in Equation 22). Since `SOBER` yields best performance when $k = 5$ (where k is the number of the most suspicious statements to start the breadth-first search from), we use $k = 5$ to make the results more comparable, where k is the number of the most suspicious transitions to start bread-first search from. On average, when 10% of the states are examined, Argus can help the user localize 56.82% of the bugs. As presented in our previous work [15], `SOBER` is able to localize 52.31% of the bugs in Siemens benchmark when 10% of the statements are examined; while other known techniques like *Scalable Statistical Bug Isolation* (Liblit et al. [14]) and `CT` (Cleve et al. [4]) are able to localize 40.00% and 26.36% of the bugs respectively in the same benchmark. If we assume that the effort in examining x% of the total states is comparable to x% of the total statements, then Argus outperforms the best known bug localization techniques. On the Argus web site, we present discussion showing that if Argus and a statement-based scheme (e.g. `SOBER`) have the same T-score value, the user needs to examine fewer statements using Argus. The high precision of the Argus bug report and Argus' capability in generating the bug report using a single execution make it a powerful bug localization tool even if we do not need its runtime bug detection capability.

5.3 Runtime Overhead and Profile Size

Because Siemens benchmark programs are small (138 – 516 LOC), they are not suitable testbeds to measure Argus' runtime overhead in large, more realistic ap-



(a) Training and Detection Overhead (b) Argus Transition Distribution Profile Size

Fig. 5. Argus runtime overhead and transition distribution profile size on SPEC2000

plications. We use the standard SPEC2000 benchmarks to study Argus runtime overhead and the transition distribution profile size. We use all SPEC `C` programs[2] except `176.gcc`, which the `Cetus` compiler cannot properly compile. All the experiments are run on a DELL Precision 350 workstation (3.06GHz Pentium IV, 1.5G memory) running RedHat Linux 9.0 with the Intel `C/C++` compiler `icc 8.1`. All of the programs are compiled with the recommended optimization flag `icc -O2`

Figure 5 shows Argus runtime overhead in training and detection modes and the ratio of the transition distribution profile size to the original executable size. On average, Argus suffers 24.6X overhead in training mode and 13.8X overhead in detection mode. The corresponding standard deviations are 16.9 and 8.7. The detection overhead of Argus is comparable with runtime detection tools like `DIDUCE` [8], `rtcc` [23] and `SafeC` [1], and is considerably lower than tools like `Purify` [10] and runtime type checking [16]. Argus' detection scheme is compatible with random sampling [13]. It can use random sampling to achieve low runtime overhead at the cost of more detection executions. Therefore, Argus is also a good technique to be used in production runs. The transition distribution profile size is 38.6% of the original executable size on average, with a standard deviation of 28.3%.

6 Conclusions and Future Work

Statistical debugging is a powerful technique for identifying bugs that do not violate programming rules or program invariants. In this paper, we present an online statistical bug detection technique called Argus. Argus is capable of detecting bugs in a single execution and can raise an alert at runtime when bug symptoms occur. Argus eliminates the requirement for labeling passing and failing executions. Argus generates more precise bug reports than the best known bug localization techniques.

[2] `164.gzip`, `175.vpr`, `177.mesa`, `179.art`, `181.mcf`, `183.equake`, `186.crafty`, `188.ammp`, `197.parser`, `254.gap`, `255.vortex`, `256.bzip2`, and `300.twolf`

The authors are investigating making Argus capable of detecting bugs reflected in program behaviors other than runtime control flow. Second, we want to develop implementations for other languages (like `Java`), and to reduce the runtime overhead.

References

[1] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*, pages 290–301, 1994.

[2] L. Burnell and E. Horvitz. Structure and chance: melding logic and probability for software debugging. *Communications of the ACM*, 38(3):31–ff., 1995.

[3] G. Casella and R. L. Berger. *Statistical Inference*. Duxbury Press, second edition, 2001.

[4] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, pages 342–351, 2005.

[5] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, 2005.

[6] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 339–348, 2001.

[7] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, 2000.

[8] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, 2002.

[9] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between fault-revealing test behavior and differences in program spectra. *Journal of Software Testing, Verifications, and Reliability*, 10(3):171–194, 2000.

[10] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Winter Technical Conference*, 1992.

[11] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200, 1994.

[12] T. A. Johnson, S.-I. Lee, L. Fei, A. Basumallik, G. Upadhyaya, R. Eigenmann, and S. P. Midkiff. Experiences in using cetus for source-to-source transformations. In *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2004.

[13] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, pages 141–154, 2003.

[14] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 conference on Programming Language Design and Implementation*, 2005.

[15] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: Statistical model-based bug localization. In *Proceedings of The fifth joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 05)*, 2005.

[16] A. Loginov, S. H. Yong, S. Horwitz, and T. W. Reps. Debugging via run-time type checking. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, pages 217–232, 2001.

[17] Software errors cost U.S. economy $59.5 billion annually, 2002. NIST News Release 2002-10.

[18] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering*, pages 465–475, 2003.

[19] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. In *Proceedings of the 5th International Workshop on Automated and Algorithmic Debugging*, pages 287–296, 2003.

[20] M. Renieris and S. P. Reiss. Fault localizationwith nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 30–39, 2003.

[21] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, 1998.

[22] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 144, 2001.

[23] J. L. Steffen. Adding run-time checking to the portable C compiler. *Software Practice and Experience*, 22(4):305–316, 1992.

[24] K. Teknomo. Recursive simple statistics tutorial. online document. http://people.revoledu.com/kardi/tutorial/RecursiveStatistic/.

[25] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO'04)*, 2004.

From Faults Via Test Purposes to Test Cases: On the Fault-Based Testing of Concurrent Systems

Bernhard K. Aichernig and Carlo Corrales Delgado

International Institute for Software Technology,
United Nations University, Macau SAR China,
P.O. Box 3058, Macau
{bka, carlo}@iist.unu.edu

Abstract. Fault-based testing is a technique where testers anticipate errors in a system under test in order to assess or generate test cases. The idea is to have enough test cases capable of detecting these anticipated errors. This paper presents a theory and technique for generating fault-based test cases for concurrent systems. The novel idea is to generate test purposes from faults that have been injected into a model of the system under test. Such test purposes form a specification of a more detailed test case that can detect the injected fault. The theory is based on the notion of refinement. The technique is automated using the TGV test case generator and an equivalence checker of the CADP tools. A case study of testing web servers demonstrates the practicability of the approach.

1 Introduction

The area of specification-based testing has advanced over the last couple of years and the results contributed to a reconciliation between the testing and the formal verification communities. Testing is now commonly acknowledged as a complementary V&V technique, if carried out systematically and well-founded. Gaudel was the most prominent who started this process [1]. Since then, many techniques and tools have emerged that generate test cases from formal specifications and are based on complete and sound testing theories.

However, the field is far from being complete, as the growing number of publications in this area indicates. Non-classical testing paradigms have to be studied and incorporated into theories. The formal underpinnings allow a deeper understanding of the relationships between testing and other verification theories, like simulation and refinement. This will lead to further applications of tools, like model checkers and constraint solvers, to generate test cases.

In this paper, we present a method that aims to advance the field in the following directions. (1) Mutation testing, traditionally applied to program text is applied on the specification level. (2) A model checker is not used to generate the test cases directly, but to generate test purposes, a high-level description of the testing goal. The method is founded on the testing theories on labeled transition systems. Tool support comes from the TGV test case generator [2] as well as from the CADP tools [3]. In particular, we address the following problems.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 324–338, 2006.

Problem 1 **Lack of test selection strategy.**

The early work on conformance testing in the area of distributed systems was mainly concerned with the soundness and completeness of the testing theory. Emphasis was given to develop a realistic conformance relation and a test case generation algorithm that was sound (no false negatives) and complete (no false positives). Since the models were finite labeled transition systems (LTSs), the problem of how to select a manageable subset out of the exhaustive test set was not a major concern. Abstraction was used to cope with the complexity. This lack of a test selection strategy limited the application domain to highly abstract protocol specifications.

Problem 2 **Identifying test purposes.**

To overcome this shortcoming, test purposes have been introduced. Here, a test purpose is a special LTS that specifies the subset of test cases to be generated. With test purposes, a tester can steer a test case generator according to his strategy. However, the problem remains, how many and which test purposes to select. Thus, the problem has been lifted, but not entirely solved.

In our approach, we want to support the tester in formalising test purposes, by turning his focus on possible faults. Possible faults can be anticipated by inspecting a specification, by using domain knowledge, or by heuristic mutation operators. In all cases, the fault is modeled at the specification level by altering the specification. We call this altered version a mutant. The idea, is to generate test cases that would find such faults in the implementation.

Problem 3 **Equivalent mutants.**

A common problem in this approach is known as the Equivalent Mutant Problem. Not all mutations represent actual faults that can be observed at the interface level. Thus, no test case exists that can distinguish the original from such an equivalent mutant.

On the specification level, equivalence checkers can be used to eliminate such equivalent mutants. The problem is which equivalence relation, based on simulations, is appropriate for our purposes. Once, the equivalence relation is fixed, the problem is solved.

Problem 4 **Test generation automation.**

The technique should automatically generate test cases. Many use the counter examples (or witnesses) produced by a model checker as test cases. However, a counter example is not a test case in the traditional sense. A test case should provide the stimuli and the responses for a system. However, a counterexample exemplifies only one possible choice of computation (a path). In case of nondeterminism involved this is not sufficient for a test case, since a test case should predict and take care of all possible responses, as well as reject wrong responses.

Therefore, we propose to use the counterexample as a test purpose. A test case generator, then, will generate a proper test case to cover the counterexample. Hence, our idea is to generate test purposes from injected faults, such that the

generated test cases will discover this anticipated faults. Fault-prevention, not structural coverage is our testing strategy.

The paper is organized as follows. After this introduction, Section 2 introduces the models, the testing theory as well as the concept of test purpose. Then, Section 3 develops the general properties of fault-based test purposes. Next, Section 4 presents the technique to generate test purposes using the CADP tools. A case study, briefly summarized in Section 5 completes the picture. Finally, we draw the conclusions and discuss related work in Section 6.

2 Conformance Testing

In this section we introduce the models for test case generation and explain how they are used to describe specifications, implementations, test cases and test purposes. These models are based on the classical formalism of labelled transition systems (LTSs) with distinguished inputs and outputs. For a full definition of the testing theory we refer to [4].

2.1 Input-Output Conformance

Definition 1 (Input-Output LTS). *An IOLTS is an LTS* $M{=}(Q^M, A^M, \rightarrow_M, q_0^M)$ *with* Q^M *a finite set of states,* A^M *a finite alphabet (the labels) partitioned into three disjoint sets* $A^M = A_I^M \cup A_O^M \cup I^M$ *where* A_I^M *and* A_O^M *are respectively input and output alphabets and* I^M *is an alphabet of unobservable, internal actions,* $\rightarrow_M \subset Q^M \times A^M \times Q^M$ *is the transition relation and* q_0^M *is the initial state.*

We will use the following classical notations of LTSs for IOLTSs. Let q, q', $q_{(i)} \in Q^M, Q \subseteq Q^M, a_{(i)} \in A_I^M \cup A_O^M, \tau_{(i)} \in I^M$, and $\sigma \in (A_I^M \cup A_O^M)^*$. $q \stackrel{\epsilon}{\Rightarrow} q' =_{df} (q = q' \vee q \stackrel{\tau_1 \ldots \tau}{\rightarrow}_M q')$ and $q \stackrel{a}{\Rightarrow} q' =_{df} \exists q_1, q_2 : q \stackrel{\epsilon}{\Rightarrow}_M q_1 \stackrel{a}{\rightarrow}_M q_2 \stackrel{\epsilon}{\Rightarrow}_M q'$ which generalizes to $q \stackrel{a_1 \ldots a}{\Rightarrow} q' =_{df} \exists q_0, \ldots, q_n : q = q_0 \stackrel{a_1}{\Rightarrow}_M q_1 \ldots q_{n-1} \stackrel{a}{\rightarrow}_M q_n = q'$. We denote q **after**$_M$ $\sigma =_{df} \{q' |\ q \stackrel{\sigma}{\Rightarrow}_M q'\}$ and Q **after**$_M$ $\sigma =_{df} \bigcup_{q \in Q} (q$ **after**$_M$ $\sigma)$. We define $Out_M(q) =_{df} \{a \in A_O^M |\ q \stackrel{a}{\rightarrow}_M\}$ and $Out_M(Q) =_{df} \{Out_M(q) | q \in Q\}$. We will not always distinguish between an IOLTS and its initial state and write $M \Rightarrow_M$ instead of $q_0^M \Rightarrow_M$. We will omit the subscript $_M$ (and superscript ^{M}) when it is clear from the context.

A specification is given in a formal description language which semantics allows to describe the behavior of the specification by an IOLTS (e.g. CSP, Estelle, SDL or LOTOS). The testing assumption is that the behavior of the implementation under test (IUT) can also be described by an IOLTS which can never refuse an input.

Definition 2 (Conformance). *The conformance relation says that an IUT conforms to S iff after a trace of S, outputs of the IUT are outputs of S:*

$$IUT \textbf{ ioconf } S =_{df} \forall \sigma \in Trace(S) : Out(IUT \textbf{ after}_{IUT} \sigma) \subseteq Out(S \textbf{ after}_S \sigma)$$

Note that this is a simplified version of **ioco** [4] excluding quiescence for the sake of clarity. All results apply to **ioco** as well.

2.2 Test Purposes

Test cases for complex concurrent systems correspond to elaborate executable programs: testing a certain procedure may require (1) to initialize a set of test processes that collaborate, (2) to execute a given preamble before being able to call the procedure, (3) to run an oracle process giving a verdict if the test has passed, and (4) to execute a postamble to get the system under test into a safe state after the test has been performed. In the telecom industry, TTCN [5] a special language for expressing test cases is used. It includes classical elements of programming languages: data types, variables, control structures and procedures.

In order to cope with the complexity of real test cases, test purposes serve to specify the goals of a test. Hence, a test purpose is a specification of a test case capturing the essence of a test in a short and abstract description. In conformance testing the notion of test purpose has been standardized [6]:

Definition 3 (Test purpose, informal). *A description of a precise goal of the test case, in terms of exercising a particular execution path or verifying the compliance with a specific requirement.*

For generating test cases from test purposes, the notion of test purpose has been formalized, and implemented in tools like SAMSTAG [7], TGV [2], TorX [8], and most recently in Microsoft's XRT [9]. Here, we use the formalization of TGV.

Definition 4 (Test purpose, formal). *Given a specification S in form of an IOLTS, a test purpose is a deterministic IOLTS $TP = (Q^{TP}, A^{TP}, \rightarrow_{TP}, q_0^{TP})$ equipped with two sets of sink states, $Accept^{TP}$ which defines Pass verdicts and $Refuse^{TP}$ which allows to limit the exploration of the graph S. Furthermore, $A^{TP} = A^S$ and TP is complete ($\forall q \in Q^{TP}, a \in A^{TP} : q \overset{a}{\rightarrow}_{TP}$).*

The specification to be covered by the test cases is formed by the synchronous product of the specification S and the test purpose TP. Furthermore, as test generation only considers the observable behavior of S it can be simplified by replacing all internal actions by τ, reducing the τ actions, and determinizing the result. This reduced specification SP_{VIS} is equipped with $Accept^{VIS}$ and $Refuse^{VIS}$ sink states derived from the test purpose.

2.3 Test Cases

In this testing framework for concurrent systems, a test case is a process running in parallel to the IUT. Hence, test cases can be modeled as an IOLTS that synchronize with the model of the IUT. TGV generates such test cases from the specification and a test purpose according to the algorithm described in [2].

Here, we only give the properties of a test case $TC = (Q^{TC}, A^{TC}, \rightarrow_{TC}, q_0^{TC})$ that has been generated from the restricted and simplified specification $SP_{VIS} = (Q^{VIS}, A^{VIS}, \rightarrow_{VIS}, q_0^{VIS})$ containing only visible actions. A test case TC has the following properties:

1. $Q^{TC} \subset Q^{VIS} \cup \{fail\}$, and $q_0^{TC} = q_0^{VIS}$,
2. $A^{TC} = A_I^{TC} \cup A_O^{TC}$ with $A_I^{TC} \subseteq A_O^{IUT}$ and $A_O^{TC} \subseteq A_I^{VIS}$ (mirror image of actions and all possible outputs of IUT considered),
3. $Pass = Accept^{VIS} \cap Q^{TC}$, $Inconc \subseteq Q^{VIS}$ and $fail$ are sink states and every state of TC except $fail$ can reach either a $Pass$ or an $Inconc$ state, $fail$ and $Inconc$ states can be reached directly only by inputs,
4. $\forall q \in Q^{TC}, \forall a \in A_I^{TC} : (\exists q' \in Inconc \cup \{fail\} : q \xrightarrow{a}_{TC} q' \Rightarrow q \xrightarrow{*} Pass)$ and $(q \xrightarrow{a}_{TC} fail \Rightarrow q \not\xrightarrow{a}_{VIS})$,
5. $\forall q \in Inconc : q \not\xrightarrow{*}_{VIS} Accept$ (Accept states cannot be reached from inconclusive states),
6. $\forall q \in Q^{TC}, \forall a \in A_O^{TC} : q \xrightarrow{a}_{TC} \Rightarrow \ \forall b \neq a : q \not\xrightarrow{b}_{TC}$ (only one output action per state).

For illustration purposes, Figure 1 shows the model of a coffee machine, with a test purpose and a resulting test case. Note that other test cases would be possible for this test purpose.

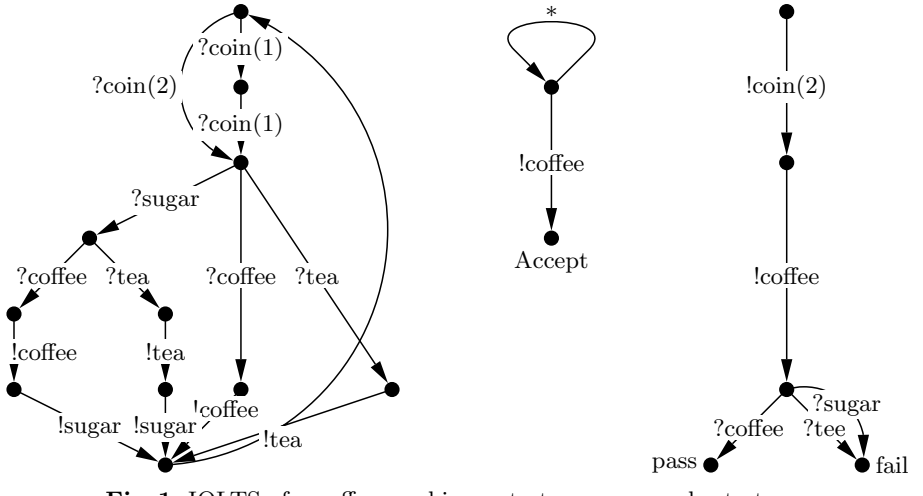


Fig. 1. IOLTS of a coffee machine, a test purpose, and a test case

3 A Theory of Fault-Based Testing

In this section, we develop a general fault-based testing theory that incorporates test purposes. As in our previous work, we use the concept of refinement as the basis to define what kind of test cases we are interested in. The difference here

is that we use the notion of refinement from [10] which relates test cases, test purposes and specifications.

3.1 Fault-Based Testing

Fault-based testing was born in practice when testers started to assess the adequacy of their test cases by first injecting faults into their programs, and then by observing if the test cases could detect these faults. This technique of mutating the source code became well-known as mutation testing and goes back to the late 70-ies [11, 12]; since then it has found many applications and has become the major assessment technique in empirical studies on new test case selection techniques [13]. In the early 90-ies formal methods entered the testing stage [14, 15] and it took not long until mutation testing was applied to formal specification languages [16]. Here, the idea is to model design errors or misinterpretations of requirements in a very early stage, and then design test cases to prevent such errors. The goal is not to test the specifications, but to derive test cases that would detect implementations of the mutated specifications. Hence, our strategy is to prevent the IUT to conform to erroneous specifications.

In our previous work, we have shown that the notion of refinement may be used to define the properties of such fault-based test cases. In [17] we discussed mutation testing in Back's refinement calculus. More recently, we developed a tool for generating test cases from mutated OCL specifications based on these ideas [18]. In the following, we will reformulate these ideas for IOLTSs and test purposes.

3.2 Relating Test Purposes, Test Cases and Specifications

In the context of software specification, programs are linked to their specification by a refinement relation. The conformance relation in Definition 2 is an example of such a refinement relation between IOLTS models. In the context of black-box testing, we can imagine a similar relation between test cases and test purposes, since test purposes are specifications of test cases [10]: The relation

$$\mathit{refines}(TC, TP, S),$$

where TC is a test case, TP a test purpose and S a specification, captures the property when a test case is a refinement of a test purpose in the context of a specification. Note that it is necessary to include the specification, since a test purpose is only meaningful together with a specification. In TGV the *refines* relation is defined by the properties of test cases presented in Section 2.3. Hence, given the test case generation algorithm of TGV $\mathit{generate}_{TGV}(TP, S)$ we have

$$\mathit{refines}(\mathit{generate}_{TGV}(TP, S), TP, S)$$

This relation expresses the fact that the test purpose TP and the specification S are consistent such that TGV is able to generate a proper test case. Hence, we abbreviate it as

$$consistent_{TGV}(TP, S) \quad =_{df} \quad refines(generate_{TGV}(TP, S), TP, S)$$

3.3 Fault-Based Test Purposes

We will now use this consistency (refinement) relation to express the property of the test purposes (test cases) we are interested in. Imagine a specification S and a mutated version S^m which has been modified by inserting a fault into S. Our goal is to generate a test case that is able to distinguish an implementation of S and S^m. For generating such a test case TC we need to formulate an appropriate test purpose TP that would guarantee that only such discriminating test cases are generated. Hence, the test purpose must not abstract away from the erroneous part in S^m, but must detect the differences between S and S^m. This can be formally expressed via the consistency relation:

$$consistent_{TGV}(TP, S) \text{ and } \neg consistent_{TGV}(TP, S^m)$$

This conjunction expresses the fact that the test purpose is able to distinguish between S and S^m, since it is inconsistent with the later. This means that the test goal can be achieved with respect to one specification, but not with the other. Consequently, a test case generated from such a test purpose TP will be able to distinguish between implementations of S and S^m. These are the test purposes we are going to generate.

One well-known challenge of mutation testing are equivalent mutants. An equivalent mutant occurs when an introduced syntactical change in the model does not represent an observable fault, hence the original S and a mutant S^m are observably equivalent ($S \approx S^m$). Without yet defining the kind of observable equivalence relation, we may formalize that if a discriminating test purpose does not exists, we have observable equivalence:

$$\nexists\, TP : (consistent_{TGV}(TP, S) \ \wedge \ \neg consistent_{TGV}(TP, S^m)) \ \Rightarrow \ (S \approx S^m)$$

This leads us immediately to the property that will be the basis for our test purpose generation:

$$(S \not\approx S^m) \ \Rightarrow \ \exists\, TP : (consistent_{TGV}(TP, S) \ \wedge \ \neg consistent_{TGV}(TP, S^m))$$

In the next section we will see that an equivalence checker for finite LTSs can be used to generate test purposes from equivalence counter examples.

4 A Technique for Fault-Based Testing

In this section, we present the technique for generating test purposes from injected faults. As indicated in the previous section, an equivalence checker for labelled transition systems forms the key technology of the generation process.

4.1 The Process

We use an equivalence checker of the CADP tools which also contain TGV. We use LOTOS as a specification language, but other input formats that can be converted to the internal CADP format for a LTS are possible.

We first present an overview of the essential steps in the process and then discuss the details.

1. Model a system to be tested in LOTOS with explicit input and output actions. Select a mutation operator for LOTOS and create a mutant L^m from the original model L.
2. Generate an IOLTS S_τ and S^m_τ from the specifications L and L^m respectively (using CADP-Caesar).
3. Simplify the rather large IOLTS S_τ and S^m_τ to obtain S and S^M using the Safety Equivalence relation (using CADP-Aldebaran).
4. Check the reduced IOLTS S and S^m for Strong Bisimulation (using CADP-Aldebaran).
5. The equivalence check gives
   (a) True: S^m is an equivalent mutant (no fault), no test purpose can be generated. Study the cause of equivalency. There might be a redundancy in the model!
   (b) False: CADP-Aldebaran issues a diagnosis (counterexample): a discriminating sequence c.
6. Add one more valid transition from S to the counterexample c (if any) in order to create a valid path which can discover the injected error. This sequence forms the wanted test purpose.
7. Generate a test case from the discriminating test purpose (using CADP-TGV).
8. Test the IUT with this test case to prevent that the IUT conforms to the faulty specification L^m.
9. Repeat this for every interesting mutation possible.

The IOLTS generated from LOTOS needs to be simplified, since it contains many redundant internal τ actions. As mentioned in Section 2.2 the test case generation process does involve visible actions only. CADP provides a simplification tool that removes all τ actions and generates an LTS that is equivalent with respect to safety properties. This Safety Equivalence relation is defined as follows [19]:

Definition 5 (Safety Equivalence). *Let $S = (Q, A_\tau, T, q_0)$ be a Labelled Transition System and let $p, r \in Q$. Safety equivalence is defined as*

$$p \approx^{saf} r \quad =_{df} \quad p \sqsubseteq^{saf} r \wedge r \sqsubseteq^{saf} p$$

The relation $\sqsubseteq^{saf}$ may be characterized as weak simulation:

$$p \sqsubseteq^{saf} r \textbf{ iff } \forall a \in A_\tau, \forall p' : (p \xrightarrow{\tau^* a} p' \Rightarrow \exists r' \bullet (r \xrightarrow{\tau^* a} r' \wedge p' \sqsubseteq^{saf} r'))$$

Then, since all τ actions are removed, a strong bisimulation check can be applied to determine the observational (non-)equivalence of both models. In the more likely case of non-equivalence, CADP generates a sequence of actions leading to a state where both behaviors deviate. Then, a test purpose IOLTS is produced which has as its only trace this sequence plus the next discriminating action of the original. This test purpose serves to generate a test case that will distinguish an implementation of the original from an implementation of the faulty mutant. This process has to be repeated for all faults one wishes to test for. Only one fault per mutant is injected (coupling effect assumption).

4.2 Mutation Operators

At the heart of this fault-based testing technique is the set of faults that are injected into a given model. These faults represent the errors one is able to anticipate and that are to be prevented by the generated test cases. They also form the basis for the coverage criterion: For each injected fault (that can be observed) there must exist a test case in the test suite able to detect it. Consequently, the set of injected faults is critical.

A common strategy in mutation testing is to define a set of mutation operators for the language in use. A mutation operator syntactically transforms a language construct, by exchanging, deleting or adding parts of it. Once the mutation operators are defined the mutants can be generated systematically or even automatically. We have defined such a set of mutation operators for Full LOTOS (see Table 1). Most of these operators are not special to LOTOS and have been considered before. ORO, SNO, ENO, LRO, RRO, MCO, ACO, STO and ASO are taken from [20]. EDO, ESO, ERO, EIO, SOR, POR, MRO, CRO, USO, HDO and PRO are taken from [21] (mutation operators for CSP). Others, like in [22], used subsets of these. We have newly added the PSP, PRP, ESP, ERP, SSP and SRP operators which are special to LOTOS.

Mutation operators are not the only source for injected faults. This is especially true on the modeling level. In security testing known vulnerabilities might be modeled as mutations. Another source for faults are common semantic misinterpretations of requirements or of a modeling language. One might imagine a set of mutation operators for UML constructs that are ambigous. Test cases could be designed such that a common interpretation of these models is enforced.

5 Web Server Case Study

In this section we report an a case study on testing web servers that serves to demonstrate that our technique is applicable. The aim was to test the correct implementation of parts of the HTTP protocol in the Apache web server. We focused on the GET-Method responsible for retrieving pages and limited ourselves to single client-server connections.

The source for the LOTOS model was the Internet standard RFC 2616 (Request for Comments). RFC 2616 specifies the syntax of the HTTP protocol in

Table 1. Mutation Operators for Extended LOTOS

Symbol	Mutation Operators	Description
EDO	Event Drop Operator	Eliminate one of the events from the process definition
ESO	Event Swap Operator	Change order of the 2 neighbouring events
ERO	Event Replacement Operator	Replace event by other events
EIO	Event Insertion Operator	Inserts one event after each event in the process definition
SOR	Sequential Operator Replacement	Replace the sequential composition operator (enabling and disabling) and [
POR	Process Operator Replacement	Replace the operator on processes (Parallel composition general case, pure interleaving and full synchronization) ||, |[]| and |||
MRO	Message Replacement Operator	Replace the message of each communication channel with other message
CRO	Channel Replacement Operator	Replace the channel with other channels within the process definition
USO	Unobservable Sequence Operator	Change the action prefix from unobservable to observable
HDO	Hiding Delete Operator	Delete an event from hide definition
PRO	Process Replacement Operator	Replace the process name with stop or exit events
SEO	Stop and Exit interchange Operator	Interchange the Stop and Exit events
PSP	Process Swap Parameter	Change order of the two neighbouring parameters in process calls
PRP	Process Replace Parameter	Replace one parameter with other in process calls
ESP	Exit Swap Parameter	Change order of the two neighbouring parameters in Exit operator calls
ERP	Exit Replace Parameter	Replace one parameter with other in Exit operator calls
SSP	Sequential composition Swap Parameter	Change order of the two neighbouring parameters in Parameterized Sequential Composition operator calls
SRP	Sequential composition Replace Parameter	Replace one parameter with other in Parameterized Sequential Composition operator calls
ORO	Operand Replacement Operators	Replace an operand (variable or constant) by another syntactically legal operand in data type declarations
SNO	Simple Expression Negation Operators	Replace a simple expression by its negation
ENO	Expression Negation Operators	Replace an expression by its negation
LRO	Logical Operators Replacement	Replace a logical operator (, ,) by another
RRO	Relational Operators Replacement	Replace a relational operator ($, \leq, , \geq, =, \neq$ on basic types or whatever is declared in data type declarations) by any other except its opposite
MCO	Missing Condition Operators	Delete conditions from conjunctions, disjunctions and implications
ACO	Adding Condition Operators	Add conditions from conjunctions, disjunctions and implications
STO	Stuck At Operators	Replace a simple expression with 0 or 1
ASO	Associative Shift Operators	Change the association between variables

BNF and describes the semantics in natural language (English). Our model consists of two LOTOS processes, the client and the server, running in parallel. The client is issuing a request message and then waits for a response message from the server process. The request message contains three parts, each one modeled as an action: (1) a Request Line (with a Method (here GET), a URI and the HTTP-version), (2) a Request Header and (3) an optional Request Body. The Request Header facilitates conditional requests, like e.g. header If_Modified_Since supports a restricted download of pages that have been recently updated. The Response message of the server contains three parts, too: (1) a Status Line (with the HTTP-version, a Status Code and a Reason), a Response Header (with information about the web page) and a Response Body (which contains the web page in most cases).

The choice of the level of granularity of the actions is a pragmatic one. Which part of the protocol is modeled as an action depends on the actual testing strategy and how the actions are easily mapped to real interactions with the web server. For example, the three parts of the Request Line have been merged into a single action, since we were not interested in testing variations of URI's and HTTP-versions.

Given the mutation operators in Table 1, almost 1500 mutants were derived from the HTTP model. Table 2 shows that a relative high number of equivalent

Table 2. Number of generated mutants

Symbol	Mutation Operator	No.Mutants	No.Equiv Mutants
EDO	Event Drop Operator	57	0
ESO	Event Swap Operator	15	0
ERO	Event Replacement Operator	65	5
EIO	Event Insertion Operator	63	7
SOR	Sequential Operator Replacement	17	7
POR	Process Operator Replacement	5	1
MRO	Message Replacement Operator	97	0
CRO	Channel Replacement Operator	46	0
USO	Unobservable Sequence Operator	2	0
HDO	Hiding Delete Operator	0	0
PRO	Process Replacement Operator	6	0
SEO	Stop and Exit interchange Operator	45	0
PSP	Process Swap Parameter	41	10
PRP	Process Replace Parameter	59	0
ESP	Exit Swap Parameter	44	15
ERP	Exit Replace Parameter	48	0
SSP	Sequential composition Swap Parameter	10	0
SRP	Sequential composition Replace Parameter	12	0
ORO	Operand Replacement Operators	154	77
SNO	Simple Expression Negation Operators	154	77
ENO	Expression Negation Operators	78	38
LRO	Logical Operators Replacement	80	8
RRO	Relational Operators Replacement	15	0
MCO	Missing Condition Operators	41	18
ACO	Adding Condition Operators	69	27
STO	Stuck At Operators	220	30
ASO	Associative Shift Operators	48	22
	Total	1491	342

mutants was obtained, especially by the ORO and SNO operators. The reason is that the synchronized product of the GET request of the client and the more complete specification of the server makes large parts of the HTTP model redundant. For example, the server is ready to listen to all kinds of Methods, but only one (GET) is actually requested by the client.

The large number of appr. 1150 non-equivalent mutants shows that a testing with mutation operators can only be done if the testing process is completely automated. However, complete automation was not the goal of this case study. Hence, we selected about 100 interesting mutations that were partly reflecting ambiguities in the HTTP standard. From these we generated the test purposes according to the process described in the previous section and used TGV to generate the test cases.

The implementation under test was our institute's Apache Web Server 2.0.40 for Red Hat Linux with HTTP/1.1 protocol. The tests have been carried out manually via a telnet session to Port 80 of the web server's URL. This is possible since the HTTP protocol is ASCII based.

We did not expect to find major flaws in the Apache Web Server, since it has been widely used since years. However, we found a case where Apache behaves unexpectedly. As mentioned above, conditional requests can be formed by adding header fields. They serve to control the caching done by a proxy server. The combination of several such header fields is underspecified in the standard. However, on page 56 of RFC 2616 the standard says:

> "An HTTP/1.1 origin server, upon receiving a conditional request that includes both a Last-Modified date (e.g., in an If-Modified-Since or If-Unmodified-Since header field) and one or more entity tags (e.g., in an

> If- Match, If-None-Match, or If-Range header field) as cache validators, *must not* return a response status of 304 (Not Modified) unless doing so is consistent with all of the conditional header fields in the request."

Entity Tags and Last-Modifed Times are metainformations used to find out whether a cache entry is an equivalent copy of an entity. The description in the RFC is ambiguous, but it indicates that priority should be given to the If-Match, If-None-Match, and If-Range header fields. The first tests showed that the Apache developers shared our interpretation:

ID	*Header 1 satisfied?*	*Header 2 satisfied?*	*Response Status*
1	If-Match = true	If-Modified-Since = true	OK (200)
2	If-Match = true	If-Modified-Since = false	Not Modified (304)
3	If-Match = false	If-Modified-Since = true	Precondition Fail (412)
4	If-Match = false	If-Modified-Since = false	Precondition Fail (412)
5	If-Match = false	If-Unmodified-Since = true	Precondition Fail (412)
6	If-Match = false	If-Unmodified-Since = false	Precondition Fail (412)
7	If-None-Match = false	If-Unmodified-Since = false	Precondition Fail (412)

Note that test cases 3–7 respond with code 412 following the interpretation that the response of the Match header has higher priority. However, the following test cases suddenly deviate from this pattern:

ID	*Header 1 satisfied?*	*Header 2 satisfied?*	*Response Status*
8	If-None-Match = false	If-Modified-Since = false	Not Modified (304)
9	If-None-Match = false	If-Unmodified-Since = true	Not Modified (304)

This is a rather unexpected response of Apache which does not seem to be consistent with the If-Match cases.

6 Conclusions

We have presented a mutation testing technique for generating test purposes. The theory relating fault-based test purposes to mutated specifications is very similar to our previous testing theory for the refinement calculus and OCL. There, a test case t for distinguishing implementations of S and S^m had to satisfy $\mathit{refines}(S, t)$ and $\neg \mathit{refines}(S^m, t)$, with *refines* being defined via weakest preconditions (refinement calculus [17]) and implication (OCL [18]). Hence, our fault-based IOLTS theory is a further instantiation of this refinement property and demonstrates its generality.

To our present knowledge this is the first work on generating test purposes via specification mutation. However, others have worked on mutation testing on the specification level before. Most of them either focus on testing the specification or on generating test cases directly. To our present knowledge Budd and Gopal were the first [23]. They applied a set of mutation operators to specifications given in predicate calculus form. The method relies on having a working implementation generating output.

Tai and Su [24] propose algorithms for generating test cases that guarantee the detection of operator errors, but they restrict themselves to the testing of singular Boolean expressions, in which each operand is a simple Boolean variable that cannot occur more than once. Tai [25] extends this work to include the detection of Boolean operator faults, relational operator faults and a type of fault involving arithmetic expressions. However, the functions represented in the form of singular Boolean expressions constitute only a small proportion of all Boolean functions.

Stocks applied mutation testing to Z specifications [16]. He presented the criteria to generate test cases to discriminate mutants, but did not automate his approach. Woodward investigated mutation operators for algebraic specifications [26].

More recently, Simon Burton presented a fault-based test case generator for Z specifications [27]. He uses a combination of a theorem prover and a collection of constraint solvers. The theorem prover generates a disjunctive normal form, simplifies the formulas and helps in formulating different testing strategies.

Black et al. studied mutation operators using the SMV model checker [28]. However, they do not consider test purposes. A group in York has recently started to use fault-based techniques for validating their CSP models [21]. Their aim is not to generate test cases, but to study the equivalent mutants. Similar research is going on in Brazil with an emphasis on protocol specifications written in the Estelle language [29].

Wimmel and Jürjens [30] use mutation testing on specifications to extract those interaction sequences that are most likely to find security issues. This work is closest to ours, since they generate test cases for finding faults in concurrent systems.

Our approach needs further evaluation. Its efficiency compared to structural model-based testing techniques needs to be analysed. Especially, the optimal choice of mutation operators deserves our attention. The case study indicates, for example, that some mutation operators are more likely to generate equivalent mutants than others.

The presented technique is specific to the TGV test case generator and similar tools. However, the theoretical discussion of the properties of fault-based test purposes has been included to make the result more widely applicable. For example, [10] discusses test purposes for the B specification language. Consequently, a similar technique could be developed for B and other model-oriented specification languages.

References

1. Gaudel, M.: Testing can be formal, too. In: TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development, Springer-Verlag (1995) 82–96
2. Jéron, T., Morel, P.: Test Generation Derived from Model-checking. In: Computer Aided Verification (CAV'99). Volume 1633 of Lecture Notes in Computer Science., Springer (1999)

3. Fernandez, J.C., Garavel, H., Kerbrat, A., Mounier, L., Mateescu, R., Sighireanu, M.: CADP: a protocol validation and verification toolbox. In Alur, R., Henzinger, T.A., eds.: Proceedings of the Eighth International Conference on Computer Aided Verification CAV. Volume 1102., Springer Verlag (1996) 437–440
4. Tretmans, J.: Test Generation with inputs, outputs and repetitive quiescence. Software: Concepts and Tools **17** (1996) 103–120
5. ISO: ISO/IEC 9646-3: Information technology - OSI - Conformance testing methodology and framework - Part 3: The Tree and Tabular Combined Notation (TTCN). Technical report, iso.ch (1998)
6. ISO: ISO/IEC 9646-1: Information technology - OSI - Conformance testing methodology and framework - Part 1: General Concepts. Technical report, iso.ch (1994)
7. Grabowski, J., Hogrefe, D., Nahm, R.: Test case generation with test purpose specification by MSC's. In: SDL'93, the 6th SDL Forum, Elsevier Science (1993) 253–266
8. de Vries, R.G., Tretmans, J.: Towards formal test purposes. In Brinksma, E., Tretmans, J., eds.: Proceedings of the Workshop on Formal Approaches to Testing of Software (FATES'01), Aalborg, Denmark, August 25, 2001. Number NS-01-4 in BRICS Notes Series, BRICS, Department of Computer Science, University of Aarhus (2001) 61–76
9. Grieskamp, W., Tillmann, N., Campbell, C., Schulte, W., Veanes, M.: Action Machines — Towards a Framework for Model Composition, Exploration and Conformance Testing Based on Symbolic Computation. In Cai, K.Y., Ohnishi, A., Lau, M., eds.: QSIC 2005, Proceedings of the Fifth International Conference on Quality Software, Melbourne, Australia, September 19–20, 2005, IEEE Computer Society (2005) 72–79
10. Ledru, Y., du Bousquet, L., Bontron, P., Maury, O., Oriat, C., Potet, M.L.: Test purposes: adapting the notion of specification to testing. In Feather, M., Goedicke, M., eds.: Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE2001), San Diego, CA, USA, 26–29 November 2001, IEEE Computer Society (2001) 127–134
11. Hamlet, R.G.: Testing programs with the aid of a compiler. IEEE Transactions on Software Engineering **3** (1977) 279–290
12. DeMillo, R., Lipton, R., Sayward, F.: Hints on test data selection: Help for the practicing programmer. IEEE Computer **11** (1978) 34–41
13. Wong, W.E., ed.: Mutation Testing for the New Century. Kluwer Academic Publishers (2001)
14. Bernot, G., Gaudel, M.C., Marre, B.: Software testing based on formal specifications: A theory and a tool. Software Engineering Journal **6** (1991) 387–405
15. Dick, J., Faivre, A.: Automating the generation and sequencing of test cases from model-based specifications. In Woodcock, J., Larsen, P., eds.: Proceedings of FME'93: Industrial-Strength Formal Methods, International Symposium of Formal Methods Europe, April 1993, Odense, Denmark, Springer-Verlag (1993) 268–284
16. Stocks, P.A.: Applying formal methods to software testing. PhD thesis, Department of computer science, University of Queensland (1993)
17. Aichernig, B.K.: Mutation Testing in the Refinement Calculus. Formal Aspects of Computing Journal **15** (2003) 280–295
18. Aichernig, B.K., Salas, P.A.P.: Test case generation by OCL mutation and constraint solving. In Cai, K.Y., Ohnishi, A., Lau, M., eds.: QSIC 2OO5, Proceedings of the Fifth International Conference on Quality Software, Melbourne, Australia, September 19-21, 2005, IEEE Computer Society Press (2005) 64–71

19. Fernandez, J.C.: Aldébaran: A tool for verification of communicating processes. Technical report, Technical Report Spectre C14, LGJ-IMAG Grenoble (1989)
20. Black, P., Okun, V., Yesha, Y.: Mutation operators for specifications. In: Proceedings of 15th IEEE International Conference on Automated Software Engineering (ASE'00). (2000) 81–88
21. Srivatanakul, T., Clark, J., Stepney, S., Polack, F.: Challenging formal specifications by mutation: a CSP security example. In: Proceedings of APSEC 2003: 10th Asia-Pacific Software Engineering Conference, Chiang Mai, Thailand, December, 2003, IEEE (2003) 340–351
22. Offutt, J., Lee, A., Rothermel, G., Untch, R.H., Zapf, C.: An experimental determination of sufficient mutant operators. ACM Transactions on Software Engineering and Methodology **5** (1996) 99–118
23. Budd, T., Gopal, A.: Program testing by specification mutation. Comput. Lang. **10** (1985) 63–73
24. Tai, K.C., Su, H.K.: Test generation for Boolean expressions. In: Proceedings of the Eleventh Annual International Computer Software and Applications Conference (COMPSAC). (1987) 278–284
25. Tai, K.C.: Theory of fault-based predicate testing for computer programs. IEEE Transactions on Software Engineering **22** (1996) 552–562
26. Woodward, M.: Errors in algebraic specifications and an experimental mutation testing tool. Software Engineering Journal (1993) 211–224
27. Burton, S.: Automated Testing from Z Specifications. Technical Report YCS 329, Department of Computer Science, University of York (2000)
28. Black, P., Okun, V., Yesha, Y.: Mutation of model checker specifications for test generation and evaluation. In: Mutation testing for the new century. Kluwer Academic Publishers (2001) 14–20
29. de Souza, S., Fabbri, J.M.S., de Souza, W.: Mutation testing applied to Estelle specifications. Software Quality Journal **8** (1999) 285–301
30. Wimmel, G., Jürjens, J.: Specification-based test generation for security-critical systems using mutations. In George, C., Huaikou, M., eds.: Proceedings of ICFEM'02, the International Conference of Formal Engineering Methods, October 21–25, 2002, Shanghai, China. LNCS, Springer-Verlag (2002) 471–482

Automated Systematic Testing of Open Distributed Programs

Koushik Sen and Gul Agha

Department of Computer Science,
University of Illinois at Urbana-Champaign, USA
{ksen, agha}@cs.uiuc.edu

Abstract. We present an algorithm for automatic testing of distributed programs, such as Unix processes with inter-process communication, Web services, etc. Specifically, we assume that a program consists of a number of asynchronously executing concurrent processes or actors which may take data inputs and communicate using asynchronous messages. Because of the large numbers of possible data inputs as well as the asynchrony in the execution and communication, distributed programs exhibit very large numbers of potential behaviors. Our goal is two fold: to execute all reachable statements of a program, and to detect deadlock states. Specifically, our algorithm uses simultaneous concrete and symbolic execution, or *concolic execution*, to explore all distinct behaviors that may result from a program's execution given different data inputs and schedules. The key idea is as follows. We use the symbolic execution to generate data inputs that may lead to alternate behaviors. At the same time, we use the concrete execution to determine, at runtime, the partial order of events in the program's execution. This enables us to improve the efficiency of our algorithm by avoiding many tests which would result in equivalent behaviors. We describe our experience with a prototype tool that we have developed as a part of our Java program testing tool jCUTE.

1 Introduction

Open distributed programs consist of asynchronous processes which communicate with each other using asynchronous messages and which may also receive data inputs from the environment. Unix process and web services are two examples of open distributed programs. The problem of testing such programs is a difficult one because of the large number of potential behaviors that they may exhibit, both because the number of possible inputs is unbounded and because there are many possible orders of execution of distributed events.

In this paper, we focus on the problem of testing for reachability of statements in distributed programs. Determining whether a statement is reachable is, in some cases, undecidable. Our goal is to automatically and efficiently find inputs and orderings which cover a large subset of the reachable statements in a program. Note that our algorithm may also detect some deadlock states during testing. Our testing algorithm builds on two ideas: *concolic execution* and *runtime partial order reduction*.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 339–356, 2006.

Concolic testing extends symbolic execution based testing [14, 16, 17, 24, 25] as follows. In symbolic execution, a program is executed using symbolic variables in place of concrete values for inputs. Each conditional expression in the program represents a constraint that determines an *execution path*. Observe that the feasible executions of a program can be represented as a tree, where the branch points in a program are internal nodes of the tree. The goal is to explore all feasible execution paths of a program [16]. The classic approach is to use depth-first exploration of the paths by backtracking [24]. Unfortunately, for large or complex program, it is computationally intractable to precisely maintain and solve the constraints required for test generation.

Concolic testing removes the limitations of symbolic execution based testing [12, 21]. Specifically, the algorithm uses simultaneous concrete and symbolic execution, or *concolic execution*, to explore distinct behaviors that may result from a program's execution given different data inputs. The key idea is as follows. We use the symbolic execution to generate data inputs that may lead to alternate behaviors. At the same time, we use the concrete execution to guide the symbolic execution along a distinct execution path. The concrete execution is also used to simplify symbolic expression that cannot be handled by our constraint solver.

For the purpose of testing for reachability and deadlocks, the behavior of a distributed program may be defined by the partial order of events at the processes, where an event is defined as the execution of a statement by a process. Testing must account for nondeterminism in the order of events, not just indeterminacy of data inputs. Moreover, the nondeterminism in the order of events arises both from the asynchrony in scheduling of processes and the delay in message delivery. Our testing algorithm forces the computation along an execution schedule which represents a particular choice for both kinds of nondeterminism.

Two difficult problems have to be addressed by our algorithm. First, concolic testing has to incorporate efficient control of the execution schedules. We use concrete executions to not only guide the symbolic execution, but also to compute the *happens before partial order relation* [7] on the events. This relation is used to determine a distinct schedule which, in general, corresponds to a different partial order. Second, we have to track symbolic expressions and constraints across process boundaries in a distributed setting.

Note that the runtime partial reduction technique we use is more involved than the standard partial order reduction [8, 10, 18, 23]: we track both symbolic constraints and the "happens-before" relation at runtime. Moreover, our partial order reduction is dynamic as the partial order is computed at runtime. This helps us to track the partial order accurately by eliminating some of the approximations required in a static analysis technique for standard partial order reduction [11].

Because our algorithm is designed to explore execution paths of a distributed program, we term our approach *Explicit Path Model Checking*. To the best of our knowledge, our algorithm is the first to consider *both inputs and schedules* for message-passing distributed programs. While other approaches [8, 11, 13, 15] have considered testing for different schedules, they use either finite domains

or random values for the inputs. Moreover, our algorithm is always sound – any bugs that it reports are real. Our algorithm is complete only under certain assumptions – namely, when our constraint solver can handle all constraints that are generated and every execution is finite. More importantly, it can significantly increase coverage as compared to testing using random inputs.

In Section 3, we describe a simple model of distributed programs which we use in Section 5 and 6 to describe our algorithm. Essentially, the model corresponds to actors [1, 2]. This allows us to describe the algorithm independent of any particular programming language. We have implemented our algorithm as a part of the tool jCUTE [20], which we have developed to test general multithreaded programs written in Java. Section 7 describes some preliminary experiments using this tool. Note that the algorithm can also be used for C programs by extending CUTE [21] with Unix processes and IPC libraries.

2 Other Related Work

A number of approaches [8, 11, 13, 15] for testing distributed programs explore all possible distinct partial orders for fixed inputs. Specifically, the approaches in [11, 13] use static partial order reduction to avoid exploring some of the different executions that have the same partial order. A reason for redundant explorations is that there are approximations associated with static analysis. One way to address this problem is to use dynamic analysis to guarantee that exactly one interleaving from each partial order is explored [15]. The approach involves storing partial orders that have already been explored; this can become a memory bottleneck. Dynamic partial order reduction [8] removes the memory bottleneck in [15] at the cost of possibly exploring more than one interleaving for each partial order. The approach in [8] works for programs which have no data input and use persistent sets. On the other hand, our dynamic partial order reduction approach is based on the macro-step Actor semantics [2]. A clear distinction between the two approaches would require a study of the adaptation of the approach in [8] to asynchronous message-passing distributed systems. The adaptation of dynamic partial order reduction to multithreaded programs is shown in [8]. In our recent work [20] extending concolic testing to multithreaded programs, we develop a new technique for dynamic partial order reduction. We believe the technique, called *race-flipping*, can be more efficient than that in [8].

Model checking tools [6, 22] based on static analysis have been developed to detect bugs in shared memory concurrent programs. These tools employ partial order reduction techniques to reduce search space. Testing shared memory multi-threaded programs using symbolic execution [24] has been developed by extending Java Pathfinder.

A number of approaches [5, 4, 9] have been developed to explore all possible global states of program that can be inferred by observing a single execution of a distributed program. These techniques are orthogonal to our algorithm and may be combined with our testing tool to enable it to also explore all reachable global states.

3 Programming Model

In order to simplify the description of our testing approach, we define a simple asynchronous message-passing imperative concurrent language MPIL (Figure 1). MPIL extends the simple language presented in [21] with message-passing primitives. An MPIL program is a set of processes that are executed concurrently, where each process executes a sequence of statements. Processes in a program communicate by passing messages asynchronously. The semantics of the language is closely related to the actor semantics–each process implements an actor [2]. However, we assume that all executions terminate or the program has deadlocked.

$$
\begin{aligned}
P &::= p_1 : I^* : Stmt^* \parallel \ldots \parallel p \ : I^* : Stmt^* \\
I &::= v \leftarrow input() \\
Stmt &::= l : S \\
S &::= v \leftarrow e \mid \texttt{if } (v \bowtie v') \texttt{ goto } l' \mid \texttt{HALT} \mid \texttt{ERROR} \mid send(p, v) \mid receive(v) \\
&\quad \text{where } p \text{ is a process, } v, v' \text{ is a variable, } \bowtie \in \{=, \neq, <, >, \leq, \geq\} \\
e &::= c \mid v \ op \ v' \quad \text{where } op \in \{+, -, /, *, \%, \ldots\},\ c \text{ is a constant}
\end{aligned}
$$

Fig. 1. Syntax of MPIL

For brevity and simplicity, we assume that new processes are not created during an execution of a program. The extension to handle these is fairly straightforward and, in fact, our implementation handles it. Moreover, an MPIL program may receive data inputs from its environment. We assume that all such inputs are available as needed; again this assumptions simplifies the description of our algorithm: our Java programs can get data inputs at any time during an execution.[1] To further simplify our exposition we assume that an MPIL program has no pointers – in fact, we assume that all variables are of type integers. However, as in [21], our algorithm can be extended to programs with pointers and complex data structures, and this is done in the implementation.

3.1 Interleaving Semantics

We now informally describe the semantics of MPIL. Consider an MPIL program P consisting of a set of processes $\mathcal{P} = \{p_1, \ldots, p_n\}$, where p_i first gets a sequence of inputs and then executes a sequence of statements, each of which is labeled. If l is the label of a statement in some process, then $l + 1$ is the label of the next statement in that process, unless the statement labeled by l is a `HALT` or an `ERROR`. The label of the initial statement of a process p is given by l_0^p. To simplify

[1] The reason this assumption does not reduce the generality of our algorithm is easy to see. Inputs are essentially unconstrained messages. Since we test for all potential external behaviors, any values of the data inputs are possible in response to any output of the program. Thus considering values as available from the beginning of the execution does not constrain the contexts in which the program is tested.

the description of the macro-step semantics (see Section 3.2), we assume that the initial statement of each process is always of the form *receive*(v).

A program may only have variables of type integer. Variables are always local to a process; they cannot be shared among processes. A process in a program can communicate with another process by sending messages using the primitive *send*. *send*(p, v) sends the content of the variable v to the process p. In the semantics of MPIL, an execution of the statement *send*(p, v) by a process p' adds the content of v to the message queue of process p. The message queue of a process is a list of values. We will use Q_p to denote the message queue of process p, $|Q_p|$ to denote the number of elements in the queue, and $Q_p[i]$ to denote the i^{th} element in the message queue. We assume that at the beginning of execution the message queue of process p_1 contains a message with content 0. A process can receive a message by calling the primitive *receive*(v). On executing *receive*(v), a process waits if its message queue is empty. Otherwise, the process *non-deterministically* picks a message from its message queue, removes the message from the queue, assigns the content of the message to v, and continues executing the next statement. The non-determinism in picking the message models the asynchrony associated with message passing.

Before executing any statement, an MPIL program gets input using the command $v \leftarrow$*input*(). This command assigns the input data to the variable v. Observe that *input*() captures the various functions through which a program may receive data from its external environment. We assume that the execution of a command of the form $v \leftarrow$*input*() is always non-blocking.

A process is said to be *active* if it has not already executed a `HALT` or an `ERROR` statement. A process is said to be *enabled* if the process is active, and the processes' message queue is non-empty if the next statement to be executed by the process is *receive*.

The operational semantics of a program in MPIL is given using a (default) scheduler which represents the choices made in a distributed execution of a program. The pseudo-code for the default scheduler can be found in [19]. We use the term *schedule* to refer to the sequence of choices.

In the scheduler, a variable pc_p represents the program counter of the process p. For each process p, pc_p is initialized to the label of the first statement of the process p (i.e. l_0^p) and Q_p is initialized to the empty list (except for Q_1). The scheduler then starts a loop. Inside the loop, the scheduler *non-deterministically* chooses an enabled process p from the set $\mathcal{P}$. It executes the next statement of the process p, where the next statement is obtained by calling *statement_at*(pc_p). During the execution of the statement the program counter pc_p of the process p is incremented by one, unless the statement is of the form `if` p `goto` l' and the predicate p in the statement evaluates to true, in which case pc_p is set to l'. The loop of the scheduler terminates when there is no enabled process in $\mathcal{P}$. The termination of the scheduler indicates either the normal termination of a program execution, or a deadlock state (when at least one process in $\mathcal{P}$ is active).

3.2 Macro-step Semantics

As shown in [2], the interleaving semantics of MPIL presented in Section 3.1 is equivalent to the *macro-step* semantics given in the form of a *macro-step scheduler* in Figure 2. In the macro-step scheduler, the execution of a process from a *receive* statement up to the next *receive* statement takes place consecutively without interleaving with any other process. The consecutive execution of all statements of a process from a *receive* statement up to the next *receive* statement is called a *macro-step* and an execution following the macro-step semantics is called a *macro-step execution*. An execution of MPIL program can be seen as a sequence of macro-steps, where at the beginning of each macro-step, using the function *choice*, the scheduler non-deterministically picks an enabled process p to be executed next and an index *msg_id* indicating that the message $Q_p[msg\_id]$ must be consumed by the next *receive* statement. The sequence of pairs of processes and message indices chosen during a macro-step execution is called a *macro-step schedule*.

```
scheduler_macro_step(P)
  pc_1 = l_0^1; ...; pc_    = l_0  ;
  Q_1 = [0]; Q_2 = [ ]; ...; Q_    = [ ];
  for each p ∈ P initialize input variables
  while (∃p ∈ P such that p is enabled)
    (p,msg_id)=choose(P);
    s =statement_at(pc_ );
    execute_concrete(p, s,msg_id);
    s =statement_at(pc_ );
    while (p is active and s ≠ receive(v))
      execute_concrete(p, s,msg_id);
      s =statement_at(pc_ );
  if (∃p ∈ P such that p is active)
    warning "Deadlock detected";

choose(P)
  pick non-deterministically a p from P
       such that p is enabled
  pick a j non-deterministically from [1..|Q_|]
  return (p, j);
```

Fig. 2. Macro-step Scheduler for MPIL

Observe that during a macro-step execution, whenever the macro-step scheduler invokes the function *choice*, a pair of process and message index is non-deterministically picked from a set of possible choices. The set of possible choices can be formally defined as follows:

$$Choices = \{(p, j) \mid p \text{ is an enabled process in } \mathcal{P} \text{ and } 1 \leq j \leq |Q_p|\}$$

The elements of this set can be lexicographically ordered as follows. We say $(p, j) < (p', j')$ iff one of the following holds:

- The index of p is less than that of p', i.e., if i and i' are such that $p = p_i$ and $p' = p_{i'}$, then $i < i'$.
- $p = p'$ and $j < j'$.

Definition 1 (next). *Given the above ordering relation $<$ over the elements of the set Choices, we can define a function next: $Choices \cup \{(\perp, \perp)\} \rightarrow Choices \cup \{(\perp, \perp)\}$ as follows. The elements of the set Choices can be ordered using the relation $<$ to get a linear sequence. If (p, j) is an element of the sequence except the last element, $next(p, j)$ is defined as the element next to (p, j) in the sequence.*

Otherwise, if (p, j) *is the last element in the sequence, then* $next(p, j)$ *is defined as* $(\perp, \perp)$. $next(\perp, \perp)$ *is defined as the first element of the sequence.*

3.3 Execution Model

We represent the execution of a statement labeled l in a process p as the *event* (p, l), and use $e, e', e_1, \ldots$ to denote events. A macro-step execution of a distributed program can be seen as a sequence of events $\tau = e_1 e_2 \ldots e_m$, such that τ is the concatenation of a sequence of sub-sequences. Each such sub-sequence has the following property. Only the first event in the sub-sequence is a receive event and each event in the sub-sequence happens at the same process. Thus each sub-sequence represents a macro-step in the execution. Note that this definition requires that the first statement of each process is a *receive*. Let $\mathcal{E}$ be the set of all macro-step executions that can be exhibited by a program on all possible inputs and schedules. In the simple testing algorithm (Section 5), our goal will be to systematically and automatically explore all executions in $\mathcal{E}$ exactly once. Later, we will refine the algorithm to avoid exploring 'equivalent' executions as much as possible (see Section 6).

We now formally define this equivalence, based on a "happens-before" relation [7]. Given an execution of a distributed program, let E be the set of events that happened during the execution. We can define a relation $\preccurlyeq \subseteq E \times E$, called "happens-before" relation, among the events of the execution as follows:

1. $e \preccurlyeq e$,
2. $e \preccurlyeq e'$ if e and e' are events of the same process and e happens before e' in the execution,
3. $e \preccurlyeq e'$ if e is the send event of a message and e' is the receive event that consumes the message sent during the event e, and
4. $e \preccurlyeq e'$ if there is a e'' such that $e \preccurlyeq e''$ and $e'' \preccurlyeq e'$.

Thus the "happens-before" relation is a partial order relation.

Given two executions τ and τ' in $\mathcal{E}$, we say that τ and τ' are *causally equivalent*, denoted by $\tau \equiv_{\preccurlyeq} \tau'$, iff τ and τ' have the same set of events and they are linearizations of the same "happens-before" relation. We use $[\tau]_{\equiv_{\preccurlyeq}}$ to denote the set of all executions in $\mathcal{E}$ that are causally equivalent to τ.

We define *the representative set of executions* $\mathcal{E}_{\equiv} \subseteq \mathcal{E}$ as the set that contains exactly one candidate from each equivalence class $[\tau]_{\equiv_{\preccurlyeq}}$ for all $\tau \in \mathcal{E}$. Formally, $\mathcal{E}_{\equiv}$ is the set such that following properties hold: $\mathcal{E}_{\equiv} \subseteq \mathcal{E}$, $\mathcal{E} = \bigcup_{\tau \in \mathcal{E}_{\equiv}} [\tau]_{\equiv_{\preccurlyeq}}$, and for all $\tau, \tau' \in \mathcal{E}_{\equiv}$, it is the case that $\tau \not\equiv_{\preccurlyeq} \tau'$.

The following result shows that a systematic and automatic exploration of each element in $\mathcal{E}_{\equiv}$ is sufficient for testing.

Proposition 1. *If a statement is reachable in a program* P *for some input and schedule, then there exists a* $\tau \in \mathcal{E}_{\equiv}$ *such that the statement is executed in* τ.

The proof of this proposition is straight-forward. If a statement is reachable then there exists an execution τ in $\mathcal{E}$ such that the execution τ executes the statement.

By the definition of $\equiv_{\preccurlyeq}$, any execution in $[\tau]_{\equiv_{\preccurlyeq}}$ executes the statement. Hence, the execution in $\mathcal{E}_{\equiv}$ that is equivalent to τ executes the statement.

The "happens-before" relation among the events can be tracked efficiently at runtime using vector clocks [7]. A vector clock $V\colon \mathcal{P} \rightarrow \mathbb{N}$ is a map from processes to natural numbers (also known as logical time). For each process p, let us associate a vector clock denoted by VC_p with p. Let $V = max(V_1, V_2)$ iff for all $p \in \mathcal{P}$, $V(p) = max(V_1(p), V_2(p))$. Let $V \leq V'$ iff for all $p \in \mathcal{P}$, $V(p) \leq V(p')$.

At the beginning of an execution, for all p and p' in $\mathcal{P}$, let $VC_p(p') = 0$. During the execution, at every event, the vector clock of a process is updated as follows.

1. If e is a send event executed by process p, then $VC_p(p) \leftarrow VC_p(p) + 1$ and attach VC_p with the message.
2. If e is a receive event executed by process p and if V is the vector clock attached with the message received, then $VC_p \leftarrow max(V, VC_p)$. This is followed by $VC_p(p) \leftarrow VC_p(p) + 1$.

We can associate a vector clock with every event e, denoted by VC_e as follows. If e is executed by p and if VC_p is the vector clock of p just before the event e, then $VC_e = VC_p$.

Given the above update rules for vector clocks during an execution, the following theorem [4] holds:

Theorem 1. *For any two events e and e', $e \preccurlyeq e'$ iff $VC_e \leq VC_{e'}$.*

We say that two events e and e' are *independent* iff $e \not\preccurlyeq e'$ and $e' \not\preccurlyeq e$. Therefore, by Theorem 1, e and e' independent iff $VC_e \not\leq VC_{e'}$ and $VC_{e'} \not\leq VC_e$.

Since we are interested only in exploring macro-step executions, henceforth, we will use the terms execution and schedule to refer to macro-step execution and macro-step schedule, respectively.

4 An Illustrative Example

We now illustrate our testing methodology by means of the simple program in Figure 3. For brevity, we omit the first *receive* statement of process p_1 in the program. We perform testing on the program by generating inputs and schedules one by one and executing the program both concretely and symbolically on these inputs and schedules. We assume that a program executes according to the macro-step semantics described above. We represent an execution diagrammatically using a lifeline where each circle on the lifeline represents a program state and each line segment between two circles represents the execution of a statement by a process. We always label such a line segment by a pair of the form (p, l) denoting the execution of the statement labeled l by the process p. We assume that time increases from top to bottom in the diagram.

Figure 4.a shows the execution of the program on a random input and a random schedule. In the execution there are three states s_1, s_2, s_3 where the program can possibly backtrack (or continue with a different path) if we can generate a different

schedule or different input. For example, at the s_1, there are two other possible choices that the scheduler may make – it may execute p_3 by receiving the value sent by the second *send* statement of p_1 or by receiving the value sent by the third *send* statement of p_1. Similarly, at s_2, the scheduler may make another choice – it may execute p_3 by receiving the message sent by the third *send* statement of p_1. At s_3, the program may take the *then* branch of the program if the input is chosen such that it satisfies the constraint $2 * y + 1 == 4$, which is generated using the simultaneous symbolic execution and constraint solving.

```
p1 :                    p2 :                    p3 :
x ← input()                                     y ← input()
1: send(p2, 1)          1: receive(z)           1: receive(u)
2: send(p3, 4)                                  2: if (u! = 2 * y + 1) goto 4
3: send(p3, x)                                  3: ERROR
                                                4: receive(u)
```

Fig. 3. Simple Distributed Program Example

In our simple testing algorithm (described in Section 5), we generate the next input or schedule by exploring other alternatives at these backtracking points in a depth-first manner. We cannot generate an input such that, in the next execution, the program takes the *then* branch at s_3. This is because the equation $2*y+1 == 4$ is unsatisfiable assuming that y is an integer. Therefore, in the next execution we execute the program by taking the alternative scheduler choice at the s_2. The execution is shown in Figure 4.b. After this execution we try to backtrack at s_3 and generate the input $\{x = 1, y = 0\}$ by solving the constraint $x == 2 * y + 1$, which is generated during the simultaneous symbolic execution. Figure 4.c gives the third execution.

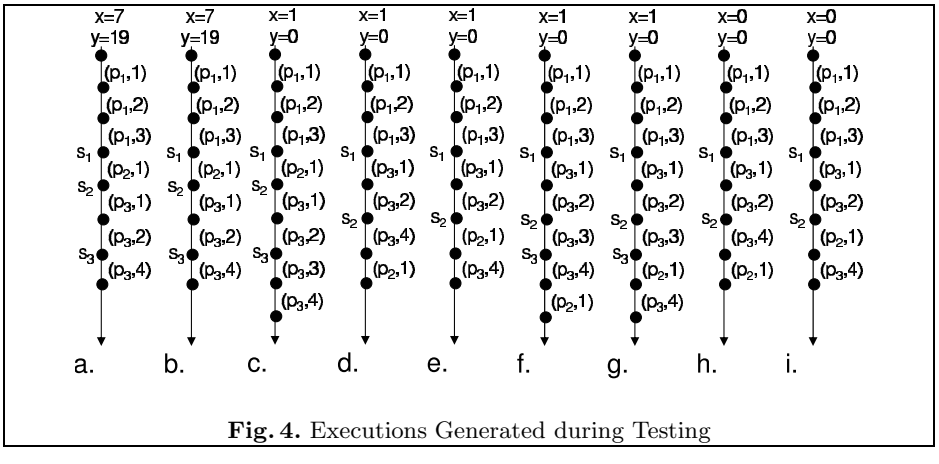


Fig. 4. Executions Generated during Testing

In this way, our simple testing algorithm proceeds in a depth-first manner either by generating an input by solving a constraint at a backtracking point or by generating different schedule by making an alternative scheduler choice

at a backtracking point. The remaining executions of the program are shown in Figure 4. Note that our simple algorithm, which considers all possible scheduler choices at a backtracking point, results in many redundant executions. We can get rid of most of the redundant executions using our efficient testing algorithm (described in Section 6) which performs *runtime partial order reduction* by computing a "happens-before" relation (described in Section 3.3) among various events in an execution.

Our efficient testing algorithm only generates the first three executions in Figure 4. This is far less than the number of executions generated by our simple testing algorithm. Our efficient testing algorithm avoided the redundant executions and yet was able to hit the error statement. In particular, at the backtracking point denoted by s_1 in Figure 4.c, our efficient algorithm does not consider the other two possible alternative choices of executing the p_3. This is because the execution of the first *receive* statement by p_2 after s_1 does not effect the execution of any future statement. Therefore, delaying the execution of the p_2 after s_1 will result in an execution that will have the same "happens-before" relation as the current execution. Considering two executions having the same "happens-before" relation is redundant since we are concerned with statement reachability (see Theorem 1).

5 Simple Algorithm

We present a simple systematic search algorithm in which our goal is to explore all macro-step execution paths of a program P by generating inputs and macro-step schedules. As in earlier work [12, 21], our algorithm uses concrete values as well as symbolic values for the inputs, and executes the program both concretely and symbolically. During the course of the execution, it collects the constraints over the symbolic values over each branching point (i.e., the *symbolic constraints*). At the end of the execution of a path, the algorithm has computed a sequence of symbolic constraints corresponding to each branching point. We call the conjunction of these constraints *path constraint*. Observe that all input values that satisfy a given path constraint will explore the same execution path given that we follow the same schedule.

The algorithm first generates a random input and a macro-step schedule which specifies the order of execution of processes. Then the algorithm does the following in a loop: it executes the code with the generated input and the schedule, and the same time records the process and message index pairs chosen by the scheduler as well as the symbolic constraints. The algorithm backtracks and alters these choices to systematically explore all possible macro-step execution paths using a depth-first search strategy. Specifically, the algorithm does one of the following to find the new data values or schedule for the next execution:

1. It picks a constraint from the symbolic constraints that were collected along the execution path and negates the constraint to define a new path constraint. The algorithm then finds some concrete values, if such values exist, that satisfy the new path constraint.

```
// input: P is the program to test
run_CUTE(P)
  completed=false; I = path_c = branch_hist=[ ];
  while not completed
    scheduler(P);
```

Fig. 5. Testing Algorithm Calls Scheduler in a Loop

2. It generates a new schedule such that at some state where the scheduler makes a choice, the *next* possible choice is picked instead of the current choice.

The algorithm continues the loop until it sweeps all feasible execution paths.

There is one complication arising from the fact that for some symbolic constraints, our constraint solver may not be powerful enough to compute concrete values that satisfy the constraints. To address this difficulty, such symbolic constraints are simplified by replacing some of the symbolic values with concrete values. Because of this, our algorithm is sound but not complete.

We now provide the details of the algorithm. The algorithm is described using a centralized interpreter for programs in MPIL. This is to simplify the description. In fact, jCUTE instruments distributed programs and uses a centralized scheduler to control the distributed processes.

The pseudo-code for our algorithm is in Figure 5. Before starting the execution loop, the algorithm initializes the logical input map $\mathcal{I}$ (which maps each input variable to a value) to an empty map [21], the sequences *path_c* (which maintains scheduler choices and symbolic constraints for a given execution), and *branch_hist* (which maintains the history of branches taken) to the empty sequences. Each element of the list *path_c* has the following fields:

1. *constraint*: stores the constraint generated on the execution of a conditional statement. At the end of an execution, the conjunction of all the constraints stored in the elements of *path_c*, for which the field *hasConstraint* is true, gives the path constraint for the given execution path. (Since in [21] we were not concerned about distributed events, each element of *path_c* was used to store only a constraint).
2. *hasConstraint*: set to true if the field *constraint* stores a constraint. It is set to false if the field *constraint* contains a scheduler choice.
3. *schedule*: stores a pair of process and message index, which is the choice made by the scheduler before executing a *receive*(v) statement.
4. *next_schedule*: stores the scheduler choice *next*(*schedule*).

The non-deterministic function *choice* given in Figure 2 is replaced by the function *choice_simple_systematic* (see Figure 6). The simple scheduler first initializes the program counters pc_p and Q_p for each process $p \in \mathcal{P}$. In addition, the simple scheduler also initializes the global counter variable i to 0. At any point of execution, i contains the sum of the number of choices made by the scheduler thus far, as well as the number of conditional statements executed. The input variables of each process are also initialized using the logical input

map $\mathcal{I}$ (cf. [21]). If $\mathcal{I}(v)$ is undefined for an input variable v, then v is initialized randomly. In the function *choose_simple_systematic*, the scheduler picks the same schedule as the previous execution as long as i is less than the number of elements of *path_c*. The list *path_c* is truncated appropriately at the end of the previous execution to perform a depth-first search of the execution paths. Otherwise, the scheduler picks a pair of process and message index such that the pair is the smallest pair in the set of possible choices.

```
scheduler(P)
  pc_{p1} = l_0^{p1}; ...; pc_{pn} = l_0^{pn};
  Q_{p1} = [0]; Q_{p2} = [ ]; ...; Q_{pn} = [ ];
  i = 0;
  for each p ∈ P initialize
      input variables using I
  while (∃p ∈ P such that p is enabled)
    (p,msg_id)=choose_simple_systematic(P);
    path_c[i].hasConstraint = false;
    i = i + 1;
    s =statement_at(pc_p);
    execute_concolic(p, s,msg_id);
    s =statement_at(pc_p);
    while (p is active and s ≠ receive(v))
      execute_concolic(p, s,msg_id);
      s =statement_at(pc_p);
  if (∃p ∈ P such that p is active)
    warning "Deadlock detected";
  compute_next_input_and_schedule();

choose_simple_systematic(P)
  if (i ≤ |path_c|)
    (p,msg_id) =path_c[i].schedule;
  else
    path_c[i].schedule = (p,msg_id) = next(⊥, ⊥);
  path_c[i].next_schedule = next(p,msg_id);
  return (p,msg_id);
```

Fig. 6. Simple Scheduler for Testing MPIL

```
execute_concolic(p, s, j)
  pc_p = pc_p + 1;
  match(s)
    case send(p', v):
      Q_{p'} = (S_p(v), A_p(v)) :: Q_{p'} ;
    case receive(v):
      (val, sval) = Q_p[j];
      S_p = S_p[v ↦ val]; A_p = A_p[v ↦ sval];
      Q_p = remove_element(Q_p, j);
    :
    :
compute_next_input_and_schedule()
  j = i − 1;
  while j ≥ 0
    if path_c[j].hasConstraint
      if (branch_hist[j].done == false)
        branch_hist[j].branch=¬branch_hist[j].branch;
        if (∃I' that satisfies neg_last(path_c[0 ... j]))
          path_c = path_c[0 ... j];
          branch_hist=branch_hist[0 ... j];
          I = I';
          return;
    else if path_c.next_schedule ≠ (⊥, ⊥)
      path_c.schedule=path_c.next_schedule;
      path_c = path_c[0 ... j];
      branch_hist=branch_hist[0 ... j];
      return;
    j = j − 1;
  if (j < 0) completed=true;
```

Fig. 7. Concolic Execution and Compute Next Schedule and Input

5.1 Computing Next Schedule and Input

The function *compute_next_input_and_schedule*, described in Figure 7, computes the schedule and the input that will direct the next program execution along an alternative execution path. It first picks an element of *path_c* from the end. If the element contains a constraint and if it is not negated before, then constraint solving is invoked to generate a new input (see [21]). Otherwise, if the element contains a scheduler choice and if not all scheduler choices at the execution point denoted by the element have been exercised, then a new schedule is generated. Specifically, if the pair (p, m) is chosen at the execution point denoted by $path\_c[j].schedule$, then $next(p, m)$, which is stored in $path\_c[j].next\_schedule$, is assigned to $path\_c[j].schedule$. In the next execution, at that particular execution point, the scheduler will pick $next(p, m)$, a choice which was not exercised before at that execution point. This ensures that in subsequent executions all the choices are selected one by one.

5.2 Concolic Execution

Concolic execution [12, 21] performs both symbolic and concrete execution of a program side by side in a cooperative way. The concolic execution technique will be important for efficiently testing distributed programs: the availability of concrete values for all memory locations in addition to the symbolic values helps us to accurately determine the partial order of a distributed execution (as described later in Section 6). Determining the partial order is important to avoid exploring redundant executions. On the other hand, the symbolic execution part of the concolic execution helps us perform symbolic execution as much as possible. This symbolic execution combined with constraint solving is essential to generate data inputs for the next execution.

The details of the procedure *execute_concolic* can be found in [21]. A brief pseudo-code of the procedure is given in Figure 7. At runtime, for each process p concolic execution maintains a symbolic state $\mathcal{A}_p$ mapping memory locations to symbolic expressions over symbolic input values in addition to the concrete state $\mathcal{S}_p$ mapping memory locations to concrete values. During concolic execution, every statement is executed concretely using the function *evaluate_concrete* and symbolically using the function *evaluate_symbolic*. In addition to performing symbolic execution, the function *evaluate_symbolic* simplifies any complex (e.g. non-linear) symbolic expressions in the symbolic state by replacing some symbolic values in the expression by their corresponding concrete values.

Note that in concolic execution, to carry out the symbolic execution, we need to track symbolic states and symbolic constraints across the process boundaries. To achieve this, both the concrete value and the symbolic value of the variable v are sent, when a process executes a statement of the form $send(p, v)$. Moreover, for each process p the message queue Q_p is modified to a list of pairs of concrete and symbolic values. An execution of the statement $send(p, v)$ by a process p' prepends a pair of the concrete and the symbolic value of the variable v to the message queue of process p.

6 Efficient Algorithm

We now provide an efficient algorithm which explores a much smaller superset of the execution paths in $\mathcal{E}_{\equiv}$. The efficient algorithm is based on the following observation. At a point where the scheduler makes a choice, often it is sufficient to consider all messages for a particular process only as possible choices by the scheduler, instead of considering all messages for all processes as possible scheduler choices. This is because, considering all messages for all processes would result in many equivalent executions.

We now characterize the case where the scheduler has to choose between messages for different processes. Consider a prefix $\tau = e_1 e_2 \ldots e_k$ of the sequence of events in an execution, such that the scheduler makes a choice after τ. Let e be the event from process p, which happens immediately after τ when the scheduler only chooses all messages for the particular process p after τ. Now if there exists an execution τ' with prefix τe such that there is a send event e' to process p,

e' appears after τe in τ', and e is independent of e', then we need to delay the execution of process p after τ such that the receive event e of process p after τ consumes the message sent by the event e'. This would give a different non-equivalent execution. Thus in such situations, it is not sufficient if the scheduler only chooses all messages of process p after τ. Rather, immediately after τ, we need to consider all messages of at least one more process other than p.

```
scheduler(P)
  pc_{p1} = l_0^{p1}; ...; pc_{pn} = l_0^{pn};
  Q_{p1} = [0]; Q_{p2} = [ ]; ...; Q_{pn} = [ ];
  i = 0;
  for each p ∈ P initialize
      input variables using I
  while (∃p ∈ P such that p is enabled)
    (p,msg_id)=choose_simple_systematic(P);
    path_c[i].hasConstraint = false;
    path_c[i].vclock = (p, VC_p);
    i = i + 1;
    s =statement_at(pc_p);
    execute_concolic(p, s,msg_id);
    s =statement_at(pc_p);
    while (p is active and s ≠ receive(v))
      if s is send(p', v)
        for all k ≤ i
            such that (p'', V) = path_c[k].vclock
            and p'' = p' and V ⋦ VC_p and VC_p ⋦ V
          path_c[k].needs_delay=true;
      execute_concolic(p, s,msg_id);
      s =statement_at(pc_p);
  if (∃p ∈ P such that p is active)
    warning "Deadlock detected";
  compute_next_input_and_schedule();
```

```
compute_next_input_and_schedule()
  j = i − 1;
  while j ≥ 0
    if path_c[j].hasConstraint
      if (branch_hist[j].done == false)
        branch_hist[j].branch=¬branch_hist[j].branch;
        if (∃I' that satisfies neg_last(path_c[0 ... j]))
          path_c = path_c[0 ... j];
          branch_hist=branch_hist[0 ... j];
          I = I';
          return;
    else if path_c[j].next_schedule ≠ (⊥, ⊥)
      (p, m) =path_c[j].schedule;
      (p', m') =path_c[j].next_schedule;
      if p = p' or path_c[j].needs_delay
        path_c[j].schedule=path_c[j].next_schedule;
        if p ≠ p'
          path_c[j].needs_delay = false;
        path_c = path_c[0 ... j];
        branch_hist=branch_hist[0 ... j];
        return;
    j = j − 1;
  if (j < 0) completed=true;
```

Fig. 8. Efficient Scheduler for Testing MPIL

Based on the above observation, we refine the simple scheduler described in Figure 6 by one (see Figure 8) that uses the "happens-before" relation to avoid exploring equivalent executions as much as possible. We assume that the scheduler maintains vector clocks with each process and that the vector clocks are updated using the procedure described in Section 3.3. We omit the vector clock update procedure from Figure 8 to keep the description simple.

In the efficient scheduler, we keep track of the vector clocks of each *receive* event. For every *send* event we check if the *send* event can synchronize with an already executed *receive* event in some alternative execution. This is done by checking the independence of the *send* event with any previously executed *receive* event. If such a check passes, then we flag the scheduler choice at the execution point just before the independent *receive* event. The flag indicates that in some future execution, just before the *receive* event, the scheduler needs to consider all messages of at least one more process.

To keep track of vector clocks and the flag, we introduce two more fields to each element of *path_c* as follows.

1. *vclock*: stores a pair (p, V), where p is the process executing the *receive* event and V is the vector clock of the event.

2. *needs_delay*: stores the flag whose truth indicates that at the current execution point, the scheduler needs to consider all messages of more than one process. If the flag is false, then the scheduler only considers all messages of a single process.

Soundness of our algorithm is trivial. A bug reported by our algorithm is an actual bug because our algorithm provides a concrete input and schedule on which the program exhibits the bug. Moreover, our algorithm can be complete in some cases.

Proposition 2. ***(Completeness)*** *During testing a program with our efficient algorithm, if the following conditions hold:*

- *The algorithm terminates.*
- *The algorithm makes no approximation during concolic execution and it is able to solve any constraint which is satisfiable.*

then our algorithm has executed all executions in $\mathcal{E}_{\equiv}$ and we have hit all reachable statements of the program.

The proof of this proposition, while fairly intuitive, is beyond the scope of this paper. Next, we show that the efficient algorithm explores significantly fewer execution paths than the simple algorithm while achieving the same branch coverage.

7 Implementation and Experiments

We have implemented both the simple and the efficient testing algorithm as a part of the Java testing tool jCUTE. The tool can be applied to test distributed Java programs written in the Actor language [2]. The Actor language extends Java by supporting actors or processes. The language is supported as a library in Java. In the language we assume that *Java threads are not explicitly used by the programmer.*

We report our experience of using jCUTE on a few examples, which include implementations of a *leader election* algorithm, a *distributed sorting algorithm*, and *Chandy-Misra's shortest path algorithm.* We performed all experiments on a Windows XP laptop with a 2.0 GHz Pentium M processor and 1GB RAM. The tool and the code for the case studies can be downloaded from `http://osl.cs.uiuc.edu/~ksen/cute/`.

The leader election algorithm that we considered works on a system with N processes connected using a unidirectional ring. Each process is assumed to have an unique id. We considered a general implementation where we assumed that the unique ids can be any value – in fact, they are assumed to be inputs. Such a general implementation cannot be handled by the model-checker in [3].

In the implementation, when we did not assume that the communication channels are FIFO, then our testing algorithms discovered an assertion violation that shows that there can be inputs and schedules where the algorithm fails to elect a leader.

Table 1. Results of Testing Distributed Programs

		Simple Testing Algorithm			Efficient Testing Algorithm			
Name	# of Processes	Run time in seconds	# of Executions	% Branch Coverage	Run time in seconds	# of Iterations	% Branch Coverage	Bug(s) Found
Leader	3	25.1	387	66.7	0.53	9	66.7	0
Election (FIFO)	4	> 33000	> 30000	66.7	15.92	22	66.7	0
Leader (non-FIFO)	3	0.16	5	70.0	0.24	5	70.0	1
Distributed	4	0.39	14	71.43	0.21	7	71.43	0
Sorting	5	13.3	420	71.43	1.13	35	71.43	0
	6	2152.42	64636	71.43	7.63	226	71.43	0
Chandy-	4	> 2600	> 100000	62.5	8.92	338	62.5	0
Misra	5	> 2690	> 100000	62.5	15.01	562	62.5	0

When we assumed that the communication channels are FIFO, both of our testing algorithms terminated without reporting any error. Table 1 gives the various statistics about this testing experiment.

Similarly, we tested implementations of a distributed sorting algorithm and Chandy Misra's shortest path computation algorithm. A model of the sorting algorithm was used for model-checking using the SPIN model-checker. However, in that experiment, they assumed a fixed sequence of numbers for sorting. Instead, we made the numbers to be sorted as inputs. This enabled us to test the algorithm not only for all schedules but also for all inputs.

The experimental results show that for the implementations that we considered, the efficient algorithm explores significantly fewer execution paths than the simple testing algorithm. On the other hand, both the algorithms attain the same branch coverage. The branch coverage in most cases is less than 100% because the implementations contain a number of assert statements that were never violated and some dead branches which cannot be taken.

8 Conclusion

We presented a new algorithm and an implementation to systematically and efficiently test distributed programs with inputs. To our best knowledge, jCUTE is the first testing tool that can automatically and exhaustively explore all non-equivalent execution paths of a distributed program with data inputs. In contrast, all previous tools [8, 11, 15] were able to test distributed programs only with a small finite domain input or with random inputs.

Acknowledgment

The first author benefited greatly from interaction with Patrice Godefroid and Nils Klarlund during a summer internship. We would like to thank Timo Latvala, Darko Marinov, Abhay Vardhan, and Mahesh Viswanathan for providing valuable comments. This work is supported in part by the ONR Grant N00014-02-1-0715, the NSF Grant NSF CNS 05-09321, and the Motorola Grant Motorola RPF #23.

References

1. G. Agha. *Actors: A Model of Concurrent Computation*. MIT Press, 1986.
2. G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.
3. R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Computer Aided Verification*, pages 340–351, 1997.
4. O. Babaoglu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In *Distributed Systems*, pages 55–96. 1993.
5. K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.
6. M. B. Dwyer, J. Hatcliff, Robby, and V. P. Ranganath. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *Form. Methods Syst. Des.*, 25(2–3):199–240, 2004.
7. C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the Workshop on Parallel and Distributed Debugging (WPDD)*, pages 183–194. ACM, 1988.
8. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. of the 32nd Symposium on Principles of Programming Languages (POPL'05)*, pages 110–121, 2005.
9. V. K. Garg and C. M. Chase. Distributed algorithms for detecting conjunctive predicates. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, page 423. IEEE, 1995.
10. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. 1996.
11. P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *24th ACM Symposium on Principles of Programming Languages)*, pages 174–186, 1997.
12. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, 2005.
13. G. J. Holzmann. The model checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
14. J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.
15. Y. Lei and K.-C. Tai. Efficient reachability testing of asynchronous message-passing programs. In *8th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 35–, 2002.
16. G. J. Myers. *The Art of Software Testing*. Wiley, 1979.
17. C. S. Pasareanu, M. B. Dwyer, and W. Visser. Finding feasible counter-examples when model checking abstracted Java programs. In *Proc. of TACAS'01*, pages 284–298, 2001.
18. D. Peled. All from one, one for all: on model checking using representatives. In *5th Conference on Computer Aided Verification*, pages 409–423, 1993.
19. K. Sen and G. Agha. Automated systematic testing of open distributed programs. Technical Report UIUCDCS-R-2005-2647, UIUC, 2005.
20. K. Sen and G. Agha. Concolic testing of multithreaded programs and its application to testing security protocols. Technical Report UIUCDCS-R-2006-2676, UIUC, 2006.

21. K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*. ACM, 2005.
22. S. D. Stoller. Model-Checking Multi-Threaded Distributed Java Programs. In *Proc. of SPIN'00: SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 224–244. Springer, 2000.
23. A. Valmari. Stubborn sets for reduced state space generation. In *10th International Conference on Applications and Theory of Petri Nets*, pages 491–515, 1991.
24. W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, 2004.
25. T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. of the Tools and Algorithms for the Construction and Analysis of Systems*, 2005.

Formal Simulation and Analysis of the CASH Scheduling Algorithm in Real-Time Maude

Peter Csaba Ölveczky[1,2] and Marco Caccamo[1]

[1] Department of Computer Science, University of Illinois at Urbana-Champaign
[2] Department of Informatics, University of Oslo
peterol@ifi.uio.no
mcaccamo@cs.uiuc.edu

Abstract. This paper describes the application of the Real-Time Maude tool to the formal specification and analysis of the CASH scheduling algorithm and its suggested modifications. The CASH algorithm is a sophisticated state-of-the-art scheduling algorithm with advanced capacity sharing features for reusing unused execution budgets. Because the number of elements in the queue of unused resources can grow beyond any bound, the CASH algorithm poses challenges to its formal specification and analysis. Real-Time Maude extends the rewriting logic tool Maude to support formal specification and analysis of object-based real-time systems. It emphasizes generality of specification and supports a spectrum of analysis methods, including symbolic simulation and (unbounded and time-bounded) reachability analysis and LTL model checking. We show how we have used Real-Time Maude to experiment with different design modifications of the CASH algorithm using both Monte Carlo simulation and reachability analysis. We could quickly and easily specify and analyze these modifications using Real-Time Maude, and discovered subtle behaviors in the modifications that lead to missed deadlines.

1 Introduction

Real-Time Maude [14, 15, 16] is a high-performance tool that extends the rewriting logic-based Maude system [4, 5] to support the formal specification and analysis of object-based real-time systems. Real-Time Maude emphasizes ease and expressiveness of specification, and provides a spectrum of analysis methods, including symbolic simulation through timed rewriting, time-bounded temporal logic model checking, and time-bounded and unbounded search for reachability analysis. Real-Time Maude differs from formal real-time tools such as the timed/hybrid automaton-based tools UPPAAL [1], Kronos [19], and Hytech [7] by having a more expressive specification formalism which supports well the specification of "infinite-control" systems which cannot be specified by such automata. Real-Time Maude has proved useful for analyzing advanced communication protocols [9, 12, 17] and wireless sensor network algorithms [18].

This paper describes the application of Real-Time Maude to the formal specification and analysis of the sophisticated state-of-the-art CASH scheduling algorithm [3] developed by the second author in joint work with Buttazzo and Sha.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 357–372, 2006.

The CASH algorithm attempts to maximize system performance while guaranteeing that critical tasks are executed in a timely manner. This is achieved by maintaining a queue of unused execution budgets that can be reused by other jobs to maximize processor utilization. The second author has suggested a modification of the algorithm which may further improve its performance.

The CASH algorithm poses challenges to its formal modeling and analysis, since we discovered during Real-Time Maude execution that there is no upper bound on the number of spare budgets in the queue. This implies that finite-control formalisms, such as the above mentioned UPPAAL, Kronos, and HyTech, which do not support unbounded data types (except for real numbers), cannot model this protocol, and that standard decision procedures cannot be applied to analyze the reachable state space.

We have used Real-Time Maude to analyze the modified algorithm and some additional design alternatives before the costly effort of implementing and testing it on a real-time kernel is undertaken. Our analysis focused on the critical property that tasks do not miss their deadlines. Time-bounded reachability analysis found a subtle scenario leading to a missed deadline in the modified algorithm. We also describe how we subjected the scheduling algorithm to Monte Carlo simulation by generating jobs pseudo-randomly. Such simulation provides not only more "realistic" simulation of the protocol, but also another light-weight analysis method which covers many—but not all—possible behaviors of the system. Moreover, extensive Monte Carlo simulation indicates that the critical missed deadline would be difficult to find during traditional testing.

2 Real-Time Maude

A Real-Time Maude *timed module* specifies a *real-time rewrite theory* [13] of the form (Σ, E, IR, TR), where:

- (Σ, E) is a *membership equational logic* [10] theory with Σ a signature[1] and E a set of conditional equations. The theory (Σ, E) specifies the system's state space as an algebraic data type. (Σ, E) must contain a specification of a sort `Time` modeling the time domain (which may be dense or discrete).
- IR is a collection of *labeled conditional instantaneous rewrite rules* specifying the system's *instantaneous* (i.e., zero-time) local transitions, each of which is written `crl [`l`] :` t `=>` t' `if` *cond*, where l is a *label*. Such a rule specifies a *one-step transition* from an instance of t to the corresponding instance of t', *provided* the condition holds. The rewrite rules are applied *modulo* the equations E.[2]
- TR is a set of *tick (rewrite) rules*, written with syntax

[1] i.e., Σ is a set of declarations of *sorts*, *subsorts*, and *function symbols* (or *operators*)

[2] The set E of equations is a union $E' \cup A$, where A is a set of equational axioms such as associativity, commutativity, and identity, so that deduction is performed *modulo* A. Operationally, a term is reduced to its E'-normal form modulo A before any rewrite rule is applied.

`crl [`l`] : {`t`} => {`t'`} in time` τ `if` *cond* `.`

that model the elapse of time in a system. `{_}` is a built-in constructor of sort `GlobalSystem`, and τ is a term of sort `Time` that denotes the *duration* of the rewrite.

The initial states must be ground terms of sort `GlobalSystem` and must be reducible to terms of the form `{`t`}` using the equations in the specifications. The form of the tick rules then ensures uniform time elapse in all parts of the system.

In object-oriented Real-Time Maude modules, a *class* declaration

`class` C `|` att_1 `:` s_1`, ... ,` att_n `:` s_n `.`

declares a class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C in a given state is represented as a term $< O : C \mid att_1 : val_1, ..., att_n : val_n >$ where O is the object's *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n. In a concurrent object-oriented system, the state, which is usually called a *configuration*, is a term of the built-in sort `Configuration`. It has typically the structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative, so that rewriting is *multiset rewriting* supported directly in Real-Time Maude. The dynamic behavior of concurrent object systems is axiomatized by specifying each of its concurrent transition patterns by a rewrite rule. For example, the rule

```
rl [l] :  < O : C | a1 : x, a2 : y, a3 : z >
          < O' : C | a1 : w, a2 : 0, a3 : v >
         =>
          < O : C | a1 : x + w, a2 : y, a3 : z >
          < O' : C | a1 : w, a2 : x, a3 : v > .
```

defines a family of transitions where two objects of class `C` synchronize to update their attributes when the `a2` attribute of one of the objects has value `0`. The transitions have the effect of altering the attribute `a1` of the object `O` and the attribute `a2` of the object `O'`. "Irrelevant" attributes (such as `a3`, `a2` of `O`, and the *right-hand side* occurrence of `a1` of `O'`) need not be mentioned in a rule.

Timed modules are *executable* under reasonable assumptions, and Real-Time Maude provides a spectrum of analysis capabilities. We summarize below the Real-Time Maude analysis commands used in our case study.

Real-Time Maude's *timed "fair" rewrite* command simulates *one* behavior of the system *up to a certain duration*. It is written with syntax

`(tfrew` t `in time <=` τ `.)`

where t is the term to be rewritten and τ is a ground term of sort `Time`.

Real-Time Maude's timed *search* command uses a breadth-first strategy to search for states that are reachable from a given initial state t within time τ and match a *search pattern* and satisfy a *search condition*. The command which searches for *one* state satisfying the search criteria has syntax

`(tsearch [1]` t `=>*` *pattern* `such that` *cond* `in time <=` τ `.)`

The `such that`-condition may be omitted. Real-Time Maude also provides an *untimed* command to search for a state reachable in any amount of time. Such search, while not guaranteed to terminate, is sometimes more efficient than timed search since it does not have to keep track of durations.

Real-Time Maude also extends Maude's *linear temporal logic model checker* [5] to check whether each behavior "up to a certain time," as explained in [15], satisfies a temporal logic formula. Restricting the computations to their time-bounded prefixes means that properties can be model checked in specifications that do not allow *Zeno behavior*, since (assuming, e.g., discrete time) only a finite set of states can then be reached from an initial state. *State propositions*, possibly parameterized, should be declared as operators of sort `Prop`, and their semantics should be given by (possibly conditional) equations of the form

`{`*statePattern*`} |=` *prop* `=` *b*

for b a term of sort `Bool`, which defines the state proposition *prop* to hold in all states `{`t`}` where `{`t`} |=` *prop* evaluates to `true`. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as `True`, `False`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), and `U` ("until"). The time-bounded model checking command has syntax

`(mc` t `|=t` *formula* `in time <=` τ `.)`

for t the initial state and *formula* the temporal logic formula.

3 Overview of the CASH Scheduling Algorithm

In most real-time systems, schedulability of critical application tasks is guaranteed off-line by considering the tasks' *worst-case execution times* (WCETs). If the *average-case execution times* (ACETs) are significantly shorter than the WCETs, then a scheduling based on WCETs will negatively affect system performance as large amounts of processor time may remain unused. Such a waste of resources is not a good solution for those applications (the majority) in which some deadline misses can be tolerated by the system, *as long as hard tasks are guaranteed off-line*. A general technique for guaranteeing deadlines of hard activities in the presence of soft tasks with unpredictable execution times is based on the resource reservation approach [2]. Each task τ_i is served by a *constant bandwidth server* S_i that is characterized by its *maximum budget* Q_i (i.e., its allocated execution time) and its *period* T_i; hence, τ_i has a CPU reservation Q_i/T_i. Each server is scheduled according to the preemptive *earliest deadline first* (EDF) policy: at any instant of time, the CPU scheduler always chooses for execution the ready task with the earliest deadline. Using this methodology, the overall system performance becomes quite dependent on a correct resource allocation. Wrong resource assignments will result in either wasting the available resources or in lowering the tasks' responsiveness. Such a problem can be

overcome by introducing a suitable resource reclaiming technique like CASH [3], which is able to exploit early completions of some task instances to satisfy the extra execution requirements (*overruns*) of other tasks.

We give a brief overview of the CASH algorithm, which is described in detail in [3]. Tasks may be *periodic* or *aperiodic* (instances of aperiodic tasks arrive at "arbitrary" times). The idea behind the CASH algorithm is to handle overruns efficiently and increase processor utilization by reclaiming unused allocated execution times. To achieve this kind of *capacity sharing* (CASH), the system maintains a queue of unused budgets. When a task instance $\tau_{i,j}$ finishes before exhausting its capacity generated by the scheduling (i.e., its "borrowed" spare capacity plus its own maximum budget Q_i), its unused capacity, together with the deadline of $\tau_{i,j}$, is added to the CASH queue. When a server executes, it uses execution time from the spare capacities having deadlines no later the server's deadline. Only when such unused execution time is not available does it use its own allocated budget Q_i. When the system is *idle*, the spare capacity with the *earliest* deadline must be discharged according to the idling time. The following crucial result concerning off-line guarantees of schedulability is proved in [3]: Each capacity generated during the scheduling is exhausted before its deadline if and only if $\sum_{i=1}^{n} \frac{Q}{T} \leq 1$.

The CASH algorithm has been implemented in the SHARK kernel [6] to measure the performance gain and to validate the results predicted by the theory.

3.1 A Proposed Modification of the CASH Algorithm

The second author wanted to investigate if it is possible to let the system consume the budget of the spare capacity with *latest* deadline when the CPU is idling, so as not to exhaust spare capacities with earlier deadlines. Such a modification was motivated by the fact that capacities with earlier deadlines are more valuable than those with later ones; in fact, the shorter the deadline of a capacity is, the more likely it is that a task with overrun will be able to use such a capacity. This question was the starting point for our Real-Time Maude analysis: Could we experiment with the modified version of the CASH algorithm to decide whether the crucial schedulability result also holds for this modified algorithm, before embarking on the laborious tasks of proving the algorithm correct and implementing it on a real-time kernel?

4 Real-Time Maude Specification of the CASH Algorithms

We present in this section a sample (4 out of 10 rewrite rules) of the Real-Time Maude specification of the CASH algorithm and its proposed optimization for *all possible* task sets. The entire executable specification is given in [11]. We cover all possible task sets by allowing a job to arrive at *any* time and to execute for *any* non-zero amount of time. The tasks are not modeled explicitly; the arrival of a new task instance is modeled by a server becoming *active*, and the end of its execution time is modeled by the server becoming *idle*.

Since a system may have any number of task servers, we specify the CASH protocols in an object-oriented style, following the specification techniques given in [16]. A *state* of our system is a multiset, i.e., a term of sort `Configuration`, consisting of: a number of task server objects; the CASH queue of available spare capacities; and a constant `AVAILABLE-PROCESSOR` of sort `Configuration`, which is present in the state when no server is executing.

4.1 Modeling the Queue of Spare Capacities

We represent a *spare capacity* as a term `deadline:` d `budget:` b, where d is its *relative* deadline[3] and b is its remaining budget. The cash queue of spare capacities is represented by a term `[CASH:` $c_1 \ldots c_n$ `]`, where $c_1 \ldots c_n$ is a list of spare capacities. The Real-Time Maude sorts and operators for this data type are given as follows:

```
sorts Capacity CapacityQueue .     subsort Capacity < CapacityQueue .
op deadline:_budget:_ : Time Time -> Capacity [ctor] .
op emptyQueue : -> CapacityQueue [ctor] .
op __ : CapacityQueue CapacityQueue -> CapacityQueue
                                       [ctor assoc id: emptyQueue] .
sort Cash .     subsort Cash < Configuration .
op `[CASH:_`] : CapacityQueue -> Cash [ctor] .
```

A spare capacity whose relative deadline or remaining budget is `0` is removed from a queue by the following equations:

```
var T : Time .
eq deadline: T budget: 0 = emptyQueue .
eq deadline: 0 budget: T = emptyQueue .
```

We use a function `addCapacity` to add a spare capacity to a CASH queue. It is defined so that the cash queue is ordered according to increasing deadlines.

4.2 The `Server` Class

Each server S_i is characterized by its maximum budget Q_i (i.e., its allocated execution time in a period) and by its period T_i. In addition, the state of a server is given by: whether the server is *idle*, *executing* a task instance, or *waiting* to execute; its current deadline $d_{i,k}$; and its remaining budget c_i in the current period. We model each server as an object of the following class `Server`:

```
class Server |
      maxBudget      : NzTime,       --- maximum budget (Q_i)
      period         : NzTime,       --- period (T_i)
      state          : ServerState,  --- state of the server
      usedOfBudget   : Time,         --- time executed of OWN budget
```

[3] The *relative* deadline is the time remaining until the deadline.

```
      timeToDeadline : Time,          --- time until "current" deadline
      timeExecuted   : Time .         --- current job executed till now

sort ServerState .
ops idle waiting executing :  -> ServerState [ctor] .
```

The class attributes `maxBudget` and `period` denote, respectively, the server's maximum budget and its period. The attribute `usedOfBudget` gives the current value of $Q_i - c_i$, and the attribute `timeToDeadline` gives the current *relative* deadline, i.e., the time remaining until time $d_{i,k}$. It is implicit in the informal specification that each task instance must be executed for a *non-zero* amount of time. Therefore, we need the attribute `timeExecuted` to be able to ensure that each job executes for a non-zero amount of time.

4.3 The Instantaneous Transitions of the System

The *instantaneous* state changes in the CASH algorithm are:

1. An `idle` server S_i becomes *active* when a new job arrives. S_i goes into state `waiting` if another server with an earlier deadline is executing, and goes into state `executing` if the processor is available or if S_i can preempt the executing server.
2. An `executing` server can finish executing a job at any time after it has executed for a non-zero amount of time. It must also deposit any unused execution budget into the CASH queue. The waiting server, if any, with the earliest deadline should start/resume its execution.

A task instance that arrives before the server is idle can be regarded as either a continuation of the previous job, or as a new job that arrives when the server has been idle for zero time.

The following variables are used in the rules and equations below:

```
vars Si Sj : Oid .              vars C C' REST-OF-SYSTEM : Configuration .
var STATE : ServerState .       var CASH : Cash .
var BUDGET-LEFT : Bool .        var CQ : CapacityQueue .
vars T T' T'' T''' REMAINING-BUDGET : Time .
vars NZT NZT' Ti Qi : NzTime .
```

In [3], the case when a server becomes active is described as follows: *When a task instance $\tau_{i,j}$ arrives and the server is idle, the server generates a new deadline $d_{i,k} = max(r_{i,j}, d_{i,k-1}) + T_i$ and c_i is recharged at the maximum value Q_i.*

The following rewrite rule models the case where the server S_i becomes active while another server S_j is executing. In this case, S_i must update its deadline[4] and either preempt S_j and start executing, or go into state `waiting`, depending on whether S_i's new deadline (`T + Ti`) is earlier than S_j's current deadline (`T'`):

[4] The "current" time is the release time $r_{i,j}$, so this part will not contribute to the updated *relative* deadline.

```
rl [idleToActive] :
   < Si : Server | period : Ti, state : idle, timeToDeadline : T >
   < Sj : Server | state : executing, timeToDeadline : T' >
  =>
   if (T + Ti) < T' then   --- start to execute and preempt Sj
      (< Si : Server | state :  executing, timeToDeadline : T + Ti,
                       timeExecuted : 0, usedOfBudget : 0 >
       < Sj : Server | state :  waiting >)
   else
      (< Si : Server | state : waiting, timeToDeadline : T + Ti,
                       timeExecuted : 0, usedOfBudget : 0 >
       < Sj : Server | >)
   fi .
```

The following defines how to finish the execution of a job [3]: *When a task instance finishes, the next pending instance, if any, is served using the current budget and deadline. If there are no pending jobs, the server becomes idle, the residual capacity $c_i > 0$ (if any) is inserted in the CASH queue with deadline equal to the server deadline, and c_i is set equal to zero.*

The following rule models the case where at least one server is in state `waiting`. When the server S_i finishes executing it must allow the waiting server with the earliest deadline (`T''`) to resume/start its execution. To find the waiting server with the earliest deadline, the rule must grab the *entire* state of the system, which is achieved by the use of the operator `{_}`. The rule adds the residual budget (if any) to the CASH queue. We make sure that the application of this rule does not lead us to miss a potential missed deadline, by adding a condition that the server is not in a state where the remaining allocated budget is greater than the deadline:

```
crl [stopExecuting1] :
    {< Si : Server | state : executing, usedOfBudget : T, maxBudget : Qi,
                     timeToDeadline : T', timeExecuted : NZT >
     < Sj : Server | state : waiting, timeToDeadline : T'' >
     REST-OF-SYSTEM  CASH}
   =>
    {< Si : Server | state : idle,  usedOfBudget : Qi >
     < Sj : Server | state : executing >
     REST-OF-SYSTEM
     (if BUDGET-LEFT
      then addCapacity((deadline: T' budget: REMAINING-BUDGET), CASH)
      else CASH fi)}
   if REMAINING-BUDGET := Qi monus T        /\
      BUDGET-LEFT := REMAINING-BUDGET > 0   /\
      REMAINING-BUDGET <= T'                /\   --- deadline check
      T'' == nextDeadlineWaiting(< Sj : Server | >  REST-OF-SYSTEM) .
```

The function `nextDeadlineWaiting` finds the earliest relative deadline of the servers in state `waiting` (see [11] for its formal definition).

To make our analysis more convenient, we add a constant `DEADLINE-MISS` and a rule which rewrites an object whose remaining budget is larger than its relative deadline to `DEADLINE-MISS`:

```
op DEADLINE-MISS : -> Configuration [ctor] .
crl [deadlineMiss] :
    < S_i : Server | state : STATE, usedOfBudget : T,
                     timeToDeadline : T', maxBudget : Q_i >
   =>
    DEADLINE-MISS
   if (Q_i monus T) > T' /\ STATE == waiting or STATE == executing .
```

4.4 Modeling Time and Time Elapse

For scheduling algorithms we usually assume discrete time. Our specification therefore imports the built-in module `NAT-TIME-DOMAIN-WITH-INF` which defines the time domain to be the natural numbers and adds a constant `INF` (denoting ∞) of a supersort `TimeInf`. We differentiate between three cases of time elapse:

1. Time is advancing while some server is executing its own budget.
2. Time is advancing while some server is executing a spare capacity from the CASH queue.
3. Time is advancing while the system is idle, i.e., when no server is executing.

The tick rewrite rules modeling the first two cases are shown in [11]. The third case must be treated in two different ways, depending on whether we model the original specification or its proposed modification.

The CASH algorithm and its suggested modification can be defined by different modules that import the module `CASH-COMMON` which defines the common behavior of the two versions, and specify the tick rewrite rule for time elapse when the system is idling (i.e., when the constant `AVAILABLE-PROCESSOR` is present in the state). For the *original* CASH algorithm such time elapse is described as follows in [3]: *Whenever the processor becomes idle for an interval of time Δ, the capacity c_q (if exists) with the earliest deadline in the CASH queue is decreased by the same amount of time until the CASH queue becomes empty.* The following timed module defines time advance in idle systems and completes the Real-Time Maude specification of the original version of the CASH algorithm:

```
(tomod CASH-USE-EARLIEST-BUDGET-WHEN-IDLING is including CASH-COMMON .
  var SERVERS : Configuration .    var CASH : Cash .    var T : Time .
  crl [tickIdle] :
      {SERVERS  AVAILABLE-PROCESSOR  CASH}
     =>
      {delta(SERVERS, T)   AVAILABLE-PROCESSOR
       delta(useSpareCapacity(CASH, T), T)}  in time T
     if T <= mte(SERVERS) [nonexec] .
endtom)
```

The tick rule is *time-nondeterministic*, as time may advance by *any* amount `T` less than or equal to `mte(SERVERS)`. In Section 5, we analyze the system using a time sampling strategy that advances time by one time unit in each tick rule application. The function `delta` defines the effect of time elapse on server objects and on the CASH queue, and the function `mte` defines the maximum amount by which time can elapse. For example, time acts on the CASH queue by decreasing the relative deadlines of the capacities according to the elapsed time:

```
eq delta([CASH: CQ], T) = [CASH: delta(CQ, T)] .
op delta : CapacityQueue Time -> CapacityQueue .
eq delta(emptyQueue, T) = emptyQueue .
eq delta((deadline: NZT budget: NZT') CQ, T) =
     ((deadline: (NZT monus T) budget: NZT') delta(CQ, T)) .
```

The crucial function `useSpareCapacity` decreases the budget of the spare capacities, in order of their *increasing* deadlines, according to the elapsed time:

```
op useSpareCapacity : Cash Time -> Cash .
op useSpareCapacity : Cash Time Time -> Cash .
eq useSpareCapacity(CASH, T) = useSpareCapacity(CASH, T, 0) .
eq useSpareCapacity([CASH: emptyQueue], T, T') = [CASH: emptyQueue] .
eq useSpareCapacity([CASH: (deadline: NZT budget: NZT') CQ], T, T') =
     if T <= min(NZT monus T', NZT') then    --- enough time in budget
        [CASH: (deadline: NZT budget: NZT' monus T) CQ]
     else useSpareCapacity([CASH: CQ], T monus min(NZT monus T', NZT'),
                                       T' + min(NZT monus T', NZT')) fi .
```

The module which defines the *modified* CASH algorithm is entirely similar to the above module. The only difference is that the occurrence of the operator `useSpareCapacity`, which discharges budgets from the capacities with the earliest deadlines, in the above tick rule is replaced by an occurrence of the following operator `useLatestSpareCapacity`, which discharges capacities from the CASH queue (if any) with the *latest* deadlines when the system is idling:

```
op useLatestSpareCapacity : Cash Time -> Cash .
eq useLatestSpareCapacity([CASH: emptyQueue], T) = [CASH: emptyQueue] .
eq useLatestSpareCapacity([CASH: CQ (deadline: NZT budget: NZT')], T) =
     if T <= NZT' then [CASH: CQ (deadline: NZT budget: NZT' monus T)]
     else useLatestSpareCapacity([CASH: CQ], T monus NZT') fi .
```

5 Formal Analysis of the CASH Algorithms

The main purpose of our analysis is to investigate whether the schedulability result that each capacity generated during the scheduling can be exhausted before its deadline also holds for the modified version of the algorithm. That is, is it possible to reach a state where the execution of the remaining budget cannot be done within the current deadline?

We first used *timed fair rewriting* to quickly prototype the specification. This prototyping indicated that states with arbitrarily large number of spare capacities in the CASH queue, and with arbitrarily large relative deadlines, can be reached from initial states with just two or three servers. Since the reachable state space is infinite, we can use Real-Time Maude's *untimed* search command as a semi-decision procedure for the reachability problem since the desired state will eventually be found if it is reachable, and can use Real-Time Maude's *time-bounded* search (and LTL model checking) to explore all states that can be reached within a given time from the initial state. Such time-bounded analyses are decision procedures when the specification is *non-Zeno*, which is the case for the CASH algorithm when the length of each job is greater than zero.[5]

Before presenting our analysis, we summarize its main results. We defined some initial states, and selected the time sampling strategy '`def 1`' which increments time by one time unit in each application of a tick rewrite rule, so that all possible task sets can explored. Both time-bounded and, hence, untimed search were able to find states which could lead to missed deadlines in the *modified* CASH algorithm. In addition, we could exhibit the sequence of rewrite steps leading to such states, to ensure that they represent valid behaviors in the modified CASH algorithm. It is worth remarking that no special ingenuity was needed to define the initial states from which missed deadlines could be reached.

The specification has a high degree of nondeterminism, and, consequently, a large number of states can be reached in a short time. For example, more than 151,000 distinct states were encountered by the untimed search before it reached the missed deadline. It took Real-Time Maude 50 seconds (untimed search) and 140 seconds (time-bounded search) on a 3 GHz Pentium Xeon processor to find the missed deadlines in the two-server system, and 160 seconds and 360 seconds, respectively, for the three-server system.

We have also subjected the *original* CASH algorithm to a similar analysis. We used timed search to show that no missed deadline can be reached within time 14 in the two-server system.[6] Finally, we let the untimed search command execute for several hours from our initial states without finding a missed deadline in the original algorithm.

5.1 Defining Initial States

We can easily experiment with different system configurations in Real-Time Maude by defining appropriate initial states. We define below a state `init2` with two servers and a state `init5` with three servers. Since the the sum of the bandwidths of the servers in each state is less than or equal to 1, it should not be possible to reach a missed deadline from either state if the algorithm is correct:

```
ops init2 init5 : -> GlobalSystem .
```

[5] The advantage of untimed search over time-bounded search is that the former is in some cases more efficient, since it ignores the "time stamps" of the states [16].

[6] For the same initial state, a missed deadline is reachable in time 12 in the modified algorithm.

```
eq init2 =
   {< s1 : Server | maxBudget : 2, period : 5, timeExecuted : 0,
                    state : idle, usedOfBudget : 0, timeToDeadline : 0 >
    < s2 : Server | maxBudget : 4, period : 7, timeExecuted : 0,
                    state : idle, usedOfBudget : 0, timeToDeadline : 0 >
    [CASH: emptyQueue]   AVAILABLE-PROCESSOR} .

eq init5 =
   {< s1 : Server | maxBudget : 1, period : 3, state : idle, ... >
    < s2 : Server | maxBudget : 4, period : 8, state : idle, ... >
    < s3 : Server | maxBudget : 4, period : 24, state : idle, ... >
    [CASH: emptyQueue]   AVAILABLE-PROCESSOR} .
```

5.2 Prototyping the CASH Algorithms

Real-Time Maude's timed fair rewrite command can be used to simulate one behavior of the modified CASH algorithm up to, for example, time 100:[7]

```
Maude> (tfrew init2 in time <= 100 .)

Result ClockedSystem :
  {[CASH: (deadline: 6 budget: 2)   (deadline: 10 budget: 2)
          (deadline: 13 budget: 4)  (deadline: 15 budget: 2)
             ...
          (deadline: 150 budget: 2) deadline: 153 budget: 4]
    < s1 : Server | timeToDeadline : 155, ... >
    < s2 : Server | timeToDeadline : 160, ... >} in time 100
```

The large number of capacities in the CASH queue is worth noticing, as well as the fact that the system did not miss a deadline. We got similar results from other simulations of both versions of the protocol, where the number of spare capacities in the CASH queue grew with the amount of time elapsed.

5.3 Reachability Analysis of the Modified CASH Algorithm

We turn to our main task, and use time-bounded search to check whether a missed deadline can be reached from state `init2` in the modified algorithm. The pattern `{DEADLINE-MISS C:Configuration}` is matched by any state which contains the constant `DEADLINE-MISS`, since the variable `C:Configuration` will be matched by the rest of the configuration. The time-bounded search among states reachable within time `12` found a missed deadline (in 140 seconds):

```
Maude> (tsearch [1] init2 =>* {DEADLINE-MISS C:Configuration} in time <= 12 .)

Solution 1
C:Configuration <- ... ;  TIME_ELAPSED:Time <- 12
```

[7] The output of Real-Time Maude executions will be manually tabulated, and parts of the output omitted in the exposition will be replaced by '...'.

The underlying trace facilities for search commands in Maude can be used to exhibit the sequence of rewrite steps leading from state `init2` to the missed deadline. The sequence, given in [11], consists of 23 rewrite steps and is a "valid" behavior in the modified algorithm. Another way of obtaining a path to the missed deadline from Real-Time Maude is to use its time-bounded LTL model checker to check whether the property

"starting from `init2`, it is invariant that no missed deadline is detected"

holds for all behaviors up to time `12`. The model checker will return a counter-example since we know that the property does not hold. The following module defines an atomic proposition `deadlineMissed` to hold for exactly those states that are matched by the pattern `{DEADLINE-MISS REST-OF-SYSTEM:Configuration}`:

```
(tomod MODEL-CHECK-LATEST is including TIMED-MODEL-CHECKER .
  protecting TEST-CASH-USE-LATEST-BUDGET-WHEN-IDLING .
  op deadlineMissed : -> Prop [ctor] .
  eq {DEADLINE-MISS REST-OF-SYSTEM:Configuration} |= deadlineMissed = true .
endtom)
```

The following command checks whether it is invariant that the negation of `deadlineMissed` holds for each state reachable within time `12` from state `init2`:

```
Maude> (mc init2 |=t [] ~ deadlineMissed in time <= 12 .)
```

This command returns a counter-example (different from the one found by search), that is a path to a missed deadline, in 384 seconds.

Were we just "lucky" with our choice of initial state to find a missed deadline? We performed the same analysis on the three-server system `init5`, and used time-bounded search to find that a missed deadline could occur within time `9` (the search took almost 360 seconds; the untimed search took 160 seconds), and no earlier than that. On the other hand, even after hours of time-bounded and untimed search, we have *not* found a missed deadline from a state with two servers with bandwidths $\frac{2}{5}$ and $\frac{3}{5}$.

5.4 Experimenting with Other Versions of CASH

We have performed similar analyses on the *original* CASH algorithm. The untimed search command ran for several hours on the initial states `init1`, `init2`, and `init5` without reaching a missed deadline. In addition, we have shown that such a state cannot be reached from `init2` within time `14`.

We have also experimented with a restriction of the modified CASH algorithm that requires a server to stay idle until the end of its period after it has finished executing in its current period. We were able to modify our high-level Real-Time Maude specification with very little effort to experiment with this restriction of the CASH algorithms. Our reachability analysis revealed that *a missed deadline could still be reached* from state `init5` (but not from state `init2`) even in this restricted setting.

6 Monte Carlo Simulations of the CASH Algorithms

We show in this section how we can modify our specification to generate new jobs pseudo-randomly for the purpose of more "realistic" randomized simulation through timed rewriting.

We generate pseudo-random jobs by having two additional attributes in the class `Server`: an attribute `timeToJob` gives the time until the next instance of a task is released; and an attribute `leftOfJob` denotes the length of the next job if it has not started, and denotes its remaining execution time otherwise. The instantaneous rewrite rules are modified in the following way:

- A server becoming active can only take place when `timeToJob` is `0`.
- The rules modeling the end of an execution can only take place when the value of the `leftOfJob` attribute is `0`. In addition, at this time, we generate a new job with pseudo-random `timeToJob` and `leftOfJob` values.

To generate pseudo-random arrival and execution times, we use a function `random` which satisfies Knuth's criteria for "good" pseudo-random functions [8]. The state must also contain the ever-changing "seed," modeled as a term `[Seed:` n`]`, to this function.

Our specification of the CASH algorithms for Monte Carlo simulation is given in [11]. We present the modified version of the rule `stopExecuting1`, where the time until the next job is released is pseudo-randomly chosen to a value between `0` and twice the period T_i of the server, and the execution time of the next job is a value between `1` and twice the length of the server's maximum budget Q_i:[8]

```
crl [stopExecuting1] :
    {< S_i : Server | state : executing, usedOfBudget : T,
                      maxBudget : Q_i, timeToDeadline : T',
                      period : T_i, leftOfJob : 0 >
     < S_j : Server | state : waiting, timeToDeadline : T'' >
     [Seed: N]   REST-OF-SYSTEM   CASH}
   =>
    {< S_i : Server | state : idle,  usedOfBudget : Q_i,
                      timeToJob : random(N) rem (2 * T_i + 1),
                      leftOfJob : 1 + random(random(N)) rem (2 * Q_i) >
     < S_j : Server | state : executing >  [Seed: random(random(N))]
     ...        --- the rest remains unchanged
```

The following command performs Monte Carlo simulation of the system `init2` (with initial seed `1`) up to time `25000`:

```
Maude> (tfrew init2(1) in time <= 25000 .)

Result ClockedSystem :
 {AVAILABLE-PROCESSOR   [CASH: deadline: 7 budget: 3 ]  [Seed: 5931]
  < s1 : Server | leftOfJob : 3, timeToDeadline : 1, timeToJob : 8, ... >
  < s2 : Server | leftOfJob : 4, timeToDeadline : 7, timeToJob : 14, ... >}
 in time 24998
```

[8] The new parts of the rules are given in italicized fonts.

The result looks more "normal" than the rewrite simulations in the previous specification. We have simulated different states, with different initial seeds, up to time `1000000`. We thought that sufficiently many combinations of jobs would have been created during this time to contain a scenario leading to a missed deadline. However, none of our Monte Carlo simulations reached a missed deadline. This fact indicates that the missed deadline would be hard to detect during traditional testing and simulation of the CASH algorithm, and underscores the usefulness of reachability analysis to discover subtle but critical errors.

7 Concluding Remarks

Real-Time Maude has proved effective in analyzing different design alternatives of the sophisticated state-of-the-art CASH scheduling algorithm. Due to the unbounded queues of spare capacities, the CASH algorithm cannot be modeled by the finite-control formalisms provided by the most popular formal real-time tools like UPPAAL and, hence, cannot be analyzed by well known *decision procedures* for the reachability problem. Using Real-Time Maude, we have instead subjected the specifications to the following spectrum of analysis methods:

1. Fair timed rewriting executions.
2. Monte Carlo simulation.
3. Untimed and time-bounded search reachability analysis.
4. Time-bounded LTL model checking.

Time-bounded search and model checking are decision procedures for the corresponding *time-bounded* properties, while *unbounded* search is a *semi-decision* procedure for the (unbounded) reachability problem.

Using methods (3) and (4) we easily discovered that the modified algorithm could not guarantee that deadlines were not missed. However, the scenarios leading to the missed deadlines were subtle and were not discovered during use of methods (1) and (2). We could experiment with different designs with much less effort than required by implementing them on real-time kernels or performing traditional testing. Moreover, extensive Monte Carlo simulations suggested that it is highly unlikely that traditional testing would have found the critical error.

The analysis methods presented analyze the system from *single* initial states and, furthermore, cannot be used to show that an undesired state can *not* be reached from the initial state. Our analysis methods can therefore only be used to search for errors or to increase our confidence in the correctness of the specification. To *prove* correctness for *all possible* inputs, *theorem provers* are needed.

The analysis reported in this paper has focused on evaluating the correctness of the designs. We should in the future also develop techniques to evaluate the performance of scheduling algorithms.

Acknowledgments. We are grateful to the anonymous referees and José Meseguer for many helpful comments on earlier versions of this paper. Partial

support of this research by ONR Grant N00014-02-1-0715, by NSF Grant CCR-0234524, by NSF Grant CCR-0237884, and by the Research Council of Norway is gratefully acknowledged.

References

1. G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In *Proc. SFM-RT 2004*, volume 3185 of *LNCS*. Springer, 2004.
2. G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency*. Springer, 2005.
3. M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proc. IEEE Real-Time Systems Symposium, Orlando*, December 2000.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.
5. M. Clavel, F. Dúran, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.2)*, December 2005. `http://maude.cs.uiuc.edu`.
6. P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A new kernel approach for modular real-time systems development. In *ECRTS'01*. IEEE, 2001.
7. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.
8. D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, second edition, 1981.
9. E. Lien. Formal modelling and analysis of the NORM multicast protocol using Real-Time Maude. Master's thesis, Dept. of Linguistics, University of Oslo, 2004.
10. J. Meseguer. Membership algebra as a logical framework for equational specification. In *Proc. WADT'97*, volume 1376 of *LNCS*. Springer, 1998.
11. P. C. Ölveczky. Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude (extended version). `http://www.ifi.uio.no/RealTimeMaude/CASH`.
12. P. C. Ölveczky, M. Keaton, J. Meseguer, C. Talcott, and S. Zabele. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. In *Proc. FASE 2001*, volume 2029 of *LNCS*. Springer, 2001.
13. P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.
14. P. C. Ölveczky and J. Meseguer. Specification and analysis of real-time systems using Real-Time Maude. In *FASE 2004*, volume 2984 of *LNCS*. Springer, 2004.
15. P. C. Ölveczky and J. Meseguer. Real-Time Maude 2.1. *Electronic Notes in Theoretical Computer Science*, 117:285–314, 2005.
16. P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 2006. To appear.
17. P. C. Ölveczky, J. Meseguer, and C. L. Talcott. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude, 2004. `http://www.ifi.uio.no/RealTimeMaude`.
18. S. Thorvaldsen and P. C. Ölveczky. Formal modeling and analysis of the OGDC wireless sensor network algorithm in Real-Time Maude. Manuscript. `http://www.ifi.uio.no/RealTimeMaude/OGDC`, October 2005.
19. S. Yovine. Kronos: A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1–2):123–133, 1997.

JAG: JML Annotation Generation for Verifying Temporal Properties*

Alain Giorgetti and Julien Groslambert

Université of Franche-Comté - LIFC,
16 route de Gray - 25030 Besancon cedex France
{giorgetti, groslambert}@lifc.univ-fcomte.fr

Abstract. We present a tool for verifying temporal properties on Java/JML classes by generating automatically JML annotations that ensure the verification of the temporal properties.

1 Introduction

We present JAG, a JML (Java Modeling Language) annotation generator for verifying temporal properties. JAG consists of a translator that transforms formulae expressed in a temporal language dedicated to Java - first introduced in [7] - into JML annotations that ensure the satisfaction of the temporal formulae.

JML (Java Modeling Language) [5] is a specification language for Java developped (by G.T. Leavens) at IOWA State University. JML annotations are introduced as Java comments using the key character '`@`'. The main annotations are `invariant`, `constraint`, `requires` and `ensures`. An `invariant` clause defines a property that must be satisfied in all visible states of the class, i.e., states before the invocation or after the termination of a method. An history `constraint` relates the value of the current state and the one of the pre-state denoted with the key word `\old`. Methods are described with preconditions (`requires`) andpostconditions (`ensures`). JML allows to declare specification variables (`ghost`) which can be assigned using a `set` clause.

The JML temporal logic extension [7] is inspired by Dwyers *Specification Pattern* [4]. It can deal with exceptional termination of methods and can express both *safety* and *liveness properties*. The semantics of the temporal formulae and the translation rules are given in details in [7] for *safety properties* and in [1] for *liveness properties*.

Take the example of a buffered transaction system (Fig. 1) encoded in Java, with a method `beginTransaction()`, which starts a new transaction, two methods `commitTransaction()` and `abortTransaction()` to respectively validate and abort (rollback) the current transaction and a `modify()` method which writes the modification in a buffer. We would like to verify on this Java class the following security properties describing the behavior of the class: (i) the buffer must be empty before beginning a new transaction and (ii) each started transaction must terminate.

* Research partially funded by the french ACI *GECCOO*.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 373–376, 2006.

```
package example.transacSystem;
public class TransactionSystem {

//@ ghost boolean trDepth = false;
//@ ghost int bufferFree;
//@ ghost int max = 100;
//@ invariant bufferFree <= max;
//@ constraint max == \old(max);
...
```

```
/*@ private  normal_behavior
  @ requires trDepth == false;
  @ ensures trDepth == true;
  @*/
public void beginTransaction() {
  //@ set trDepth = true;
  ...
};
}
```

Fig. 1. A part of a JML specification: a Buffered Transaction System

The first property is a *safety* property ("something wrong must not happen"). The second one is a *liveness* property ("under certain conditions, something good must inevitably happen").

These properties can be encoded as restrictions on infinite Java execution sequences. However, it is not easy to translate them directly to JML annotations. Therefore, we propose to designers a compact temporal logic language to express such properties, and an automatic translation into standard JML annotations that are directly inserted into the Java code under verification.

The properties (i) and (ii) can be easily expressed in the temporal logic language of [7] by the following (`bufferFree` is the variable counting the free space of the buffer):

```
(i)  after commitTransaction() normal, abortTransaction() normal
     \always {bufferFree == max}
     \unless beginTransaction() called;
(ii) after beginTransaction() called \always true \until
     abortTransaction() called, commitTransaction() called
     under_invariant {true} variant {bufferFree};
```

The first formula means that `after` a transaction is finished - when `commitTransaction()` or `abortTransaction()` terminates `normally` – and unless (`\unless`) a new transaction starts (`beginTransaction() called`), the buffer must always (`\always`) be empty. The second formula means that `after` the start of a transaction, the transaction must inevitably be (`until`) finished by a commit or a rollback. The second formula is completed with a `variant` clause which is a Java expression returning a natural number. This variant must decrease each time a method is called until the method `commitTransaction()` or `abortTransaction()` is invoked. Notice also that there is a `under_invariant` keyword, here set to `true`, that allows to define a local invariant, that permits to express an extra hypothesis for the liveness proof.

2 Description of the Tool

The JAG tool parses a Java file - possibly already JML annotated - with the Iowa State University JML tools parser and takes as other input a file containing temporal formulae (Fig. 2).

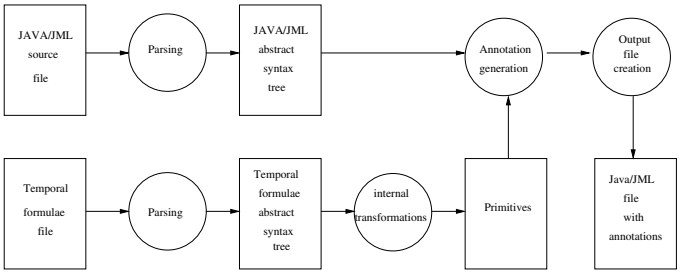


Fig. 2. Internal Structure of the Tool

Translating Temporal Formulae into Intermediate Primitives. The tool reduces each temporal property into one or more intermediate primitives that are semantically equivalent [7, 1]. The Inv primitive represents the safety part of a property. The Loop primitive represents the liveness part of a property. The Witness primitive represents special past marker on the class (for example to know if a method has already been called during the execution).

Translating Intermediate Primitives into Standard JML Annotations. Each Inv primitive is translated into a JML `invariant`. Each Loop is translated into a set of `invariant`s and history `constraint`s that imply the decrease of the variant and the deadlockfreeness of the system. Each Witness is translated as a JML `ghost` variable.

The tool generates an output file including the original file enriched with the generated JML annotations. This file can be used with other JML tools [2] to validate or prove the temporal formulae.

Trace Preservation. The tool is able to keep the trace of the generated annotations, *i.e.* it is possible, given a generated annotation, to find the original intermediate primitive and the original temporal property.

3 Experimental Results

The tool has been used on several examples. Table 1 summarizes the results obtained with the JACK [3] tool as back-end theorem prover.

Table 1. Results

Example Name	Number of temporal properties to verify	Number of line annotation generated	Number of PO (automatically proved)
TransactionSystem	2	18	92 (91)
AtmTransaction	2	21	171 (171)

4 Conclusion

The JAG tool generates JML annotations that imply the satisfaction of temporal properties (both liveness and safety) of the language defined in [7]. The particularity of this work is that the annotations are standard and can be used with all the tools taking JML files as an input. We first plan a better integration of our tool into some back-end tools and second to extend our work to other specification input, like PLTL formulae. JAG can be downloaded from the following page: `http://lifc.univ-fcomte.fr/~groslambert/JAG`.

References

1. F. Bellegarde, J. Groslambert, M. Huisman, J. Julliand, and O. Kouchnarenko. Verification of liveness properties with JML. Technical Report RR-5331, INRIA, 2004.
2. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. In Th. Arts and W. Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *ENTCS*, pages 73–89. Elsevier, 2003.
3. L. Burdy, A. Requet, and J.-L. Lanet. Java Applet Correctness: a Developer-Oriented Approach. In *Formal Methods (FME'03)*, number 2805 in LNCS, pages 422–439. Springer, 2003.
4. M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *International Conf. on Software Engineering*, pages 411–420. IEEE Computer Society Press/ACM Press, 1999.
5. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. Technical report, Iowa State University, Dept. of Computer Science, 1998.
6. G. Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1980.
7. K. Trentelman and M. Huisman. Extending JML Specifications with Temporal Logic. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology And Software Technology (AMAST'02)*, number 2422 in LNCS, pages 334–348. Springer, 2002.

LearnLib: A Library for Automata Learning and Experimentation

Harald Raffelt and Bernhard Steffen

University of Dortmund,
Chair of Programming Systems,
Baroper Str. 301,
Dortmund 44227, Germany

Abstract. In this tool demonstration we present the LearnLib, a library for automata learning and experimentation. Its modular structure allows users to configure their tailored learning scenarios, which exploit specific properties of the envisioned applications. As has been shown earlier, exploiting application-specific structural features enables optimizations that may lead to performance gains of several orders of magnitude, a necessary precondition to make automata learning applicable to realistic scenarios.

The demonstration of the LearnLib will include the extrapolation of a behavioral model for a realistic (legacy) system, and the statistical analysis of different variants of automata learning algorithms on the basis of random generated models.

1 Motivation

Most systems in use today lack adequate specifications or make use of un(der) specified components. In fact, the much propagated component-based hard- and software design style naturally leads to under specified systems, as most libraries and third party components only provide very partial specifications. To improve this situation automata learning techniques [1] have been proposed. They enable the automatic construction and subsequent update of behavioral models.

2 Automata Learning

Automata learning tries to automatically construct a finite automaton that matches the behavior of a given target automaton on the basis of (systematic) observation [2, 3]. This complements other automatic learning techniques which aim at the construction of invariants [4, 5]. The interested reader may refer to [1, 6, 7] for our (practice-oriented) view of the use of (active) learning.

Active learning assumes an omniscient teacher which is able to answer *membership* and *equivalence* queries. A membership query tests whether a string (a potential run) is contained in the target automatons language (its set of runs), and an equivalence query compares the hypothesis automaton with the target

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 377–380, 2006.

automaton for language equivalence, in order to determine whether the learning procedure was successfully completed.

In any practical attempt of learning legacy systems, equivalence queries can only be treated approximately, but membership queries can be answered by testing the target systems [1, 6]. The LearnLib therefore provides a number of heuristics and techniques for this approximation.

3 The LearnLib

The LearnLib [8] provides a platform for experimenting with different learning algorithms and for statistically analyzing their characteristics in terms of required number and size of e.g., membership queries, run time, and memory consumption. This concerns in particular the analysis of the impact of the various techniques for optimizations.

Besides the fine granular analysis for understanding the individual components of typical learning algorithms, the LearnLib can also be used as a fully automatic tool to systematically build finite state machine models of (specific aspects of) real world systems.

As depicted in Fig. 1 LearnLib consists of three modules:

- the *automata learning* module containing the basic learning algorithms,
- the *filters* module providing a number of strategies to reduce the number of queries, and
- the module for *approximative equivalence queries*, which is based on techniques adopted from the area of conformance testing. suites for the conjectures of the learning algorithms.

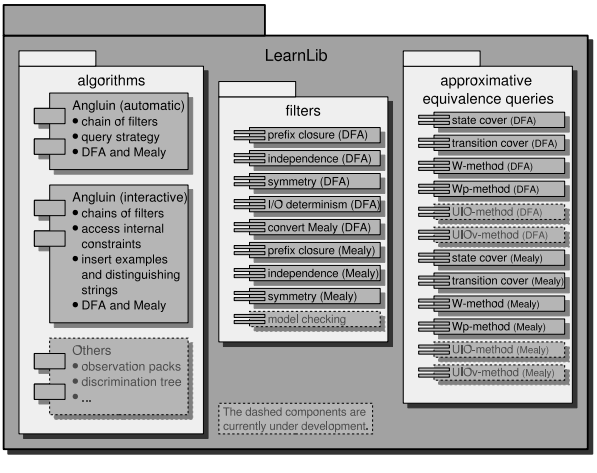


Fig. 1. LearnLib Components

3.1 Automata Learning with the LearnLib

The main module of the LearnLib contains learning algorithms of different flavor. Currently, the LearnLib only supports some variants and extensions of Angluin's algorithm L^*.

Angluin's algorithm [9] works on deterministic finite state machines, which, as usual, consist of a finite set of states Q, a finite (input) alphabet Σ, a transition function $\delta : (Q \times \Sigma) \to Q$, and a finite set of accepting states $F \subseteq Q$.

Typical reactive systems do not terminate and are thereforer not modelled appropriately with accepting states. It turned out that Mealy machines, which adequately model input/output behavior provide a much better modelling, which is not only more concise, but can also be learned much faster [7]. The LearnLib therefore provides learning algorithms for both, finite state machines and an adaptation to Mealy machines. These versions can be used in two modes: an automatic mode, which automatically proceeds in a predefined fashion, and an interactive mode, which gives the user a handle to steer the learning process in more detail.

The learning algorithms can be freely combined with a number of filters, optimizing the number of required membership queries (see [10]), and with an adequate approximate solver for equivalence queries. Whereas these approximate solvers are necessary, whenever one attempts to learn a legacy system, they can can be replaced by a perfect equivalence checker for the statistical analysis of learning scenarios using randomly generated target models. The LearnLib provides a bisimulation checker for this purpose.

3.2 Simulation and Analysis of Learning Algorithms

Different learning algorithms have very different characteristics. Perhaps most importantly, they may drastically differ in the number of membership- and equivalence queries, but they also differ in the size of their queries. Our simulator analyzed these differences, and helps to gain knowledge about how learning algorithms essentially work, and where their bottlenecks lie. The interested reader may refer to [11], where the interdependency between our optimizing filters is experimentally evaluated.

Such studies are specifically supported be the LearnLib, which, besides the configuration of individual learning scenarios, also allows the configuration of whole scenarios for the statistical analysis of learning scenarios. In fact, as will be illustrated during the demonstration, it is possible to configure a set of comparative learning scenarios and to automatically generate the corresponding comparative statistical charts.

4 Conclusion

In this paper we have presented the LearnLib, a modular library for automata learning, which is explicitly designed for experimentation. As will be illustrated

during the tool demo, its modular structure allows users to systematically analyze and then construct learning algorithms tailored for their specific application scenario.

References

1. Hagerer, A., Hungar, H., Niese, O., Steffen, B.: Model generation by moderated regular extrapolation. In R. Kutsche, H.W., ed.: FASE'02: Proc. of the 5th International Conference on Fundamental Approaches to Software Engineering. Volume 2306 of LNCS., Heidelberg, Germany, Springer-Verlag (2002) 80–95
2. Cook, J.E., Wolf, A.L.: Discovering models of software processes from event-based data. ACM Transactions on Software Engineering and Methodology (TOSEM) **7** (1998) 215–249
3. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. In Wu, J., Chanson, S.T., Gao, Q., eds.: FORTE/PSTV '99: Proc. of the Joint Int. Conference on Formal Description Techniques for Distributed System and Communication/Protocols and Protocol Specification, Testing and Verification, Kluwer Academic Publishers (1999) 225–240
4. Brun, Y., Ernst, M.D.: Finding latent code errors via machine learning over program executions. In: ICSE'04: Proc. of the 26th International Conference on Software Engineering, Edinburgh, Scotland (2004) 480–490
5. Nimmer, J.W., Ernst, M.D.: Automatic generation of program specifications. In: ISSTA'02: Proceedings of the 2002 International Symposium on Software Testing and Analysis, Rome, Italy (2002) 232–242
6. Steffen, B., Hungar, H.: Behavior-based model construction. In Mukhopadhyay, S., Zuck, L., eds.: VMCAI'03: Proc. of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation. Volume 2575 of LNCS., Springer-Verlag (2003) 5–19
7. Steffen, B., Margaria, T., Raffelt, H., Niese, O.: Efficient test-based model generation of legacy systems. In: HLDVT'04: Proc. of the 9th IEEE Int. Workshop on High Level Design Validation and Test, Sonoma (CA), USA, IEEE Computer Society Press (2004) 95–100
8. Raffelt, H., Steffen, B., Berg, T.: Learnlib: A library for automata learning and experimentation. In Halbwachs, N., Zuck, L.D., eds.: FMICS'05: Proc. of the 10th International Workshop on Formal Methods for Industrial Critical Systems. Volume 3440 of LNCS., New York, NY, USA, ACM Press (2005) 557–562
9. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **2** (1987) 87–106
10. Hungar, H., Niese, O., Steffen, B.: Domain-specific optimization in automata learning. In: Proc. of the 15th Computer Aided Verification Conference. Volume 2725 of LNCS., Springer Verlag (2003) 315–327
11. Margaria, T., Raffelt, H., Steffen, B.: Analyzing second-order effects between optimizations for system-level test-based model generation. In: ITC'05: Proc. of IEEE International Test Conference. (2005)

Trace-Based Memory Aliasing Across Program Versions

Murali Krishna Ramanathan, Suresh Jagannathan, and Ananth Grama

Department of Computer Science,
Purdue University,
West Lafayette, IN 47907
{rmk, suresh, ayg}@cs.purdue.edu

Abstract. One of the major costs of software development is associated with testing and validation of successive versions of software systems. An important problem encountered in testing and validation is memory aliasing, which involves correlation of variables across program versions. This is useful to ensure that existing invariants are preserved in newer versions and to match program execution histories. Recent work in this area has focused on trace-based techniques to better isolate affected regions. A variation of this general approach considers memory operations to generate more refined impact sets. The utility of such an approach eventually relies on the ability to effectively recognize aliases.

In this paper, we address the general memory aliasing problem and present a probabilistic trace-based technique for correlating memory locations across execution traces, and associated variables in program versions. Our approach is based on computing the *log-odds ratio*, which defines the affinity of locations based on observed patterns. As part of the aliasing process, the traces for initial test inputs are aligned without considering aliasing. From the aligned traces, the log-odds ratio of the memory locations is computed. Subsequently, aliasing is used for alignment of successive traces. Our technique can easily be extended to other applications where detecting aliasing is necessary. As a case study, we implement and use our approach in dynamic impact analysis for detecting variations across program versions. Using detailed experiments on real versions of software systems, we observe significant improvements in detection of affected regions when aliasing occurs.

1 Introduction

Identifying memory aliasing involves correlating memory locations that exhibit similar behavior across two versions of a program. Memory aliasing occurs in many software engineering applications, including impact analysis [1, 15, 17], detecting invariants [7] and correlating program properties to ensure that properties are preserved in the newer version or matching program execution histories [19]. Devising a scalable robust solution to this problem has proven to be challenging. Zhang and Gupta [19] present a relative offset based matching approach to solving the problem of matching program execution histories. While

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 381–395, 2006.

their approach performs well in the presence of simple variable renaming, it is not clear how their technique could be generalized to deal with more complex program behavior. In this paper, we present a scalable and general solution to the memory aliasing problem based on available execution traces.

Memory aliasing is the problem of identifying whether two pointers refer to the same memory location. In this paper, we address the problem of memory aliasing across program versions, i.e., given two sets `X` and `Y` of memory locations corresponding to two different versions, identify whether `x` and `y` are aliases (refer to the same memory location relatively), where $\texttt{x} \in \texttt{X}$ and $\texttt{y} \in \texttt{Y}$. We present a solution for the memory aliasing problem in the context of identifying variations across program versions. We use test results on older versions to automatically identify regions in newer versions that are affected by the changes that characterize their differences. As a first step towards detecting and isolating variations in program versions, we abstract a program as a sequence of memory reads and writes. Test inputs are fed into two versions and traces of memory operations are collected by instrumenting the binary executables. A trace is a sequence of $<$*Operation, Value*$>$ tuples, where *Operation* is either a read or write to memory and *Value* is the value read from, or written into memory. The trace is analogous to a string and the tuple analogous to an alphabet. Comparing two functions that exist in two program versions is equivalent to comparing the subsequence of the trace corresponding to the two functions under comparison.

Based on a user-defined cost function, the Levenstein [12] distance is calculated using dynamic programming and the gaps [2] in the comparison recorded. (The Levenstein distance between two strings is defined as the shortest sequence of edit operations that lead from one string to the other.) By repeating the process for multiple test inputs, cumulative information on the gaps present in the older version relative to the newer version is obtained. By projecting the tuples back to the corresponding regions in the source, information on the affected locations within an impacted function is obtained. If the Levenstein distance between the two functions is zero, then we regard the function in the newer version as unaffected by changes in the older version.

Ignoring memory locations associated with each operation in the trace and using only the operation type and associated value in calculating the Levenstein distance presents a potential problem as shown in the following example:

```
void  old() {                        void  new() {
    for(i = 0; i < 100; i++) {           for(i = 0; i < 100;i++) {
        a = i;                                   b[i] = i;
    }                                    }
}                                     }
```

Using just operation types and associated values in traces, it is easy to conclude that the functions `old` and `new` are identical, since the Levenstein distance associated with strings generated by traces on any test input is zero. However, it is obvious by examination that the functions have clearly distinct behavior.

In function `old`, values from 1...100 are written into a single memory location whereas in function `new`, the same values are written into consecutive memory locations.

To alias memory locations across versions, we rely on a model [6] used to solve a similar problem in computational biology while matching amino-acid (protein) sequences. Here, one amino acid needs to be aliased with another amino acid from which it potentially has mutated. The solution is based on computing the probability of one amino acid aligning with another amino acid in valid (known) alignments. In our approach, we execute the two versions on a test input and align the traces with respect to the operation types and the associated values (ignoring memory locations). The alignment of the memory locations for the alignments of the strings obtained using the previous step is used in computing the probability that a memory location can be aliased with any other location. This process is performed for a few test inputs and a comprehensive map of the probability that a memory location in the older version is aligned with any other location in the newer version is obtained. From this map, the memory location that has the highest probability is used as an alias. In subsequent alignment of the traces, apart from computing the equality of the operation type and values associated across versions, memory alias information computed earlier is used.

We implement our approach and evaluate its performance across a range of open-source benchmarks. In these benchmarks, the majority (approximately 90%) of the memory locations in the older version are uniquely associated with locations in the newer version. The remaining few memory locations correspond to multiple locations and these are the locations for which our approach has been found useful. When our technique is used in detecting variations between program versions, we find a significant change in the size of the impacted regions within an affected function. Furthermore, in some cases, we also find improved precision in the impact sets computed.

2 Aliasing Technique: Log-Odds Ratio

We present our aliasing technique by initially discussing a related problem in biology. Amino acid sequences of an organism's protein mutate gradually from one generation to another in the process of evolution. An important application is to determine whether two sequences are homologous or have the same ancestor. The general technique is to construct a substitution matrix where entry `(i,j)` is equal to the probability of the amino acid `i` being altered into amino acid `j` within a bounded time. There are two popular techniques to construct such a matrix in the literature: PAM [6] and BLOSSOM [9]. In this paper, we build a substitution matrix based on the technique used for PAM [6]. For our problem, we need to determine the probability that the memory location `i` in one version being the memory location `j` in another version.

We start with a discussion of the probabilistic technique used to correlate memory locations. We initially present an abstract problem in strings and then relate it to the problem of correlating memory locations that arise in software

engineering and testing environments. We are given two sets of alphabets A and B, where A ⋂ B = {}. Consider the following problem: "find a mapping from alphabets in A to alphabets in B, for all alphabets in A, such that an alphabet in A or B can have no more than a single mapping and for two strings x and y composed of alphabets in A and B respectively, the Levenstein distance of the strings x and *sub(y,x)* is minimum. Here, x is a string composed of alphabets from A, y is a string composed of alphabets from B, *sub(y,x)* is the string obtained by converting the string y by substituting each character in y by the alphabet mapped into A." For example, if we have A = {*a, b, c*}, B = {*d, e, f*}, x = *abcb* and y = *eddd*, we find that a mapping from b to d, a to e and c to f results in the smallest Levenstein distance of the strings x and *sub(y,x)*. We are unaware of a polynomial-time optimal solution to the above problem.

A relaxation of the above problem incorporates the presence of a *history*, i.e., there is a set of pairs of strings x and y, and alignment $R(x, y)$ associated with each pair. Any new pair of strings that form an input to the problem follows the historical pattern. Given this scenario, a probabilistic approach to mapping alphabets from A and B is given below. This technique has been applied in generating PAM matrices for amino acid substitutions in computational biology. For each pair of strings x and y in the history:

1. Calculate $p(a_i, b_j)$ by dividing the number of times a_i aligns opposite b_j in $R(x,y)$ by the total number of aligned pairs.
2. Calculate $p(a_i)$ by dividing the the number of times a_i appears in x by the length of x. Similarly calculate $p(b_j)$.
3. The log-odds ratio $\mathtt{LR}(a_i, b_j)$ is defined as $\log(\frac{p(a_i,b_j)}{p(a_i)p(b_j)})$

x: a1 a2 a3 a1 a1 a2 a3 a1 a1 a2 y: b4 b2 b1 b4 b3 b2 b3 b4 b4 b2

	b1	b2	b3	b4
a1	0	0	0	1
a2	0	1.74	0	0
a3	2.32	0	1.32	0

p(a1) = 0.5
p(a2) = 0.3
p(a3) = 0.2
p(b1) = 0.1
p(b2) = 0.3
p(b3) = 0.2
p(b4) = 0.4

p(a1, b4) = 0.4
p(a1, b3) = 0.1
p(a2, b2) = 0.3
p(a3, b1) = 0.1
p(a3, b3) = 0.1

Fig. 1. An example illustrating the computation of log-odds ratio

We present an illustrative example in Figure 1. A contains alphabets {a1, a2, a3} and B contains alphabets {b1, b2, b3, b4}. Strings x and y are given and are shown in the figure. The probability of occurrence of each alphabet and the joint probability of two alphabets aligning is also shown. The log-odds ratio is

calculated as specified by the formula $\frac{p(a_i,b_j)}{p(a_i)p(b_j)}$ and is specified in the table. We observe that there is perfect aliasing (when $p(a_i, a_j) = p(a_i) = p(a_j)$) between `a2` and `b2`. In contrast, there is no such aliasing between `a1` and `b4`. However, since `a1` aligns more frequently with `b4`, these two alphabets are aliased. Even though `a3` aligns with two different alphabets once, we alias it with `b1`, since it occurs only once in the string y. However, the history of pairs of alignments need to be taken into consideration to specify the alias with high confidence. We give the details below.

Compute the log-odds ratio for all possible pairs of alphabets in `A` and `B`. Iterate the process for each pair of strings in the history and obtain the cumulative log-odds ratio. On completion, for every alphabet a_i, find the alphabet b_j in `B` for which the log-odds ratio is the highest. Map the alphabet a_i to b_j, if b_j is not already mapped. Notice that $\texttt{LR}(a_i, b_j)$ can become zero, in the absence of even a single alignment between the alphabets. In practice, we observe many such pairs and these pairs can be removed immediately from consideration.

In the context of memory aliasing, the problem can be now described as follows: `A` and `B` correspond to the set of memory locations in the older version and newer version, respectively. In other words, a memory location is simply an alphabet in the appropriate set. Sequences x and y represent sequences of memory locations operated in each of the versions. To obtain the history as mentioned in the relaxed version of this problem, a *learning* process is executed. In this process, the memory locations are totally ignored and the traces, obtained by executing the versions on the same test input, are aligned using the tuple <*Operation*, *Value*>. This alignment is performed using dynamic programming. After obtaining the alignment of the trace, an alignment of the memory locations (corresponding to the tuple) across the two versions is obtained. The log-odds ratio for the locations is subsequently determined. On completion, we obtain a probabilistic aliasing of the memory locations. We elaborate on this process using the following example.

2.1 Example

We show two program fragments in Figure 2, one labeled `old`, and the other `new`. There is one memory location (`a`) associated with function `old` (local variable `i` is not considered). There are two memory locations (`b[0],b[1]`) associated with function `new`. We denote the address of any variable `v` as `mem(v)`.

Using the algorithm presented above, memory traces associated with the invocation of these functions on the same test input (`j` = 5) are first obtained. The memory trace thus generated is:

```
old: <mem(a),W,0>, <mem(a),W,1>, <mem(a),W,5>
new: <mem(b[0]),W,0>, <mem(b[1]),W,1>, <mem(b[0]),W,5>, <mem(b[1]),W,6>

Trace Element: <Location, Operation, Value>
Location: Memory location associated with the operation
```

```
void old(int j){                    void new(int j){
  for(i = 0; i < 2; i++) {            for(i = 0; i < 2; i++) {
      a = i;                              b[i] = i;
  }                                     }
  a = j;                                b[0] = j;
                                        b[1] = j + 1;
}                                   }
```

Fig. 2. Example of functions from two program versions

```
Op: Read(R),Write(W)
Value : 32 bit value
```

Initially, we ignore all memory locations and align the above strings. The gaps are represented by a hyphen(-). We obtain the following alignment:

```
old: <W, 0>, <W, 1>, <W, 5>, -
new: <W, 0>, <W, 1>, <W, 5>, <W, 6>
```

We retrieve the corresponding memory locations and get the following alignment:

```
old: mem(a), mem(a), mem(a), -
new: mem(b[0]), mem(b[1]), mem(b[0]), mem(b[1])
```

The logs-odd ratio for the memory locations are obtained using the above alignment. We get $LR(\mathtt{a}, \mathtt{b[0]})$ equals $log(4/3)$ and $LR(\mathtt{a}, \mathtt{b[1]})$ equals $log(2/3)$. By performing the above operations on other test inputs, we can correlate the memory locations of `a` and `b[0]` with high confidence. In subsequent trace alignments, we use the alias in the dynamic programming to ensure that the alphabets under comparison are identical.

3 Case Study

We discuss an implementation for discovering variations across program versions. Our analysis tool consists of two components – an instrumentation module and a comparison module. Both components operate over program binaries. The binaries, representing a program and its revision, are instrumented to record read and write operations, and execute on the same test input. The effect of the instrumentation yields memory traces on selective operations. These traces are then compared using dynamic programming, and optimally aligned (with or without memory aliasing) based on a user defined cost function. A block diagram of this process is shown in Figure 3.

Gaps in the alignment help detect operations performed by the newer version that are absent in the older version, and vice versa. Accumulating this information over all test inputs provides the set of affected regions in the newer version.

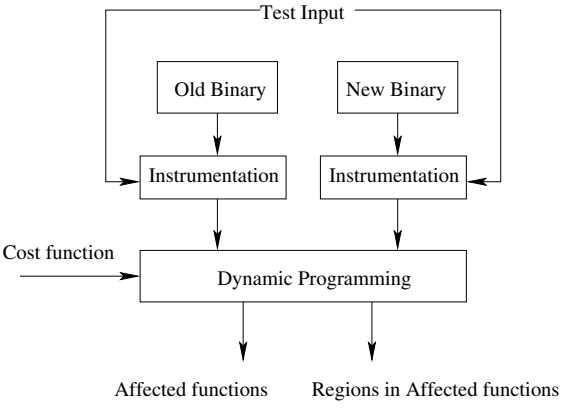


Fig. 3. Block Diagram for detecting variations across two program versions

If there are no gaps present in such a comparison over all test inputs, the functions are declared to be unaffected. Otherwise, the affected regions (in the form of line numbers) in the newer version are identified.

3.1 Instrumentation Tool Using PIN

We use PIN [16], a dynamic binary instrumentation tool, for instrumentation purposes. PIN supports a rich set of abstract operations that can be used to analyze applications at the instruction level without detailed knowledge of the underlying instruction set. Instrumentation code can be inserted at desired locations in the binary. For our current implementation, we track all heap related operations ignoring other instructions, including reads or writes to the stack.

The instrumentation module takes as input the binary and the list of functions in the binary that need to be instrumented. When the binary is executed on a given test input with dynamic instrumentation, a list of tuples is generated. The elements in the tuple include the type of operation (read or write), its 32 bit value (read or written), the corresponding memory location, the line number and the function in which the instruction was generated.

3.2 Comparison Tool Using Dynamic Programming

We provide a simple example for aligning two strings optimally. Given two strings `abacbd` and `aabcabcd`, one of the longest common subsequences is `abacd`. Table 1 presents the dynamic programming table d that gives the edit distance between the two strings. The cost at any box of the dynamic programming table d, $d_{i,j}$ is calculated as follows. If alphabets i and j are equal, then the cost $d_{i,j}$ is the minimum of $d_{i-1,j-1}$, $d_{i-1,j}+1$ and $d_{i,j-1}+1$. Otherwise, the cost $d_{i,j}$ is the minimum of $d_{i-1,j}+1$ and $d_{i,j-1}+1$. The alignment table is correspondingly update to reflect whether the diagonal, left or top element is chosen in d. The edit distance in this case is four assuming unit cost for insertions and deletions. After filling up all the values in table d, a traversal from the end of the alignment table

Table 1. Dynamic programming table d representing the gap costs

		a	a	b	c	a	b	c	d
	0	1	2	3	4	5	6	7	8
a	1	0	1	2	3	4	5	6	7
b	2	1	2	1	2	3	4	5	6
a	3	2	1	2	3	2	3	4	5
c	4	3	2	3	2	3	4	3	4
b	5	4	3	2	3	4	3	4	5
d	6	5	4	3	4	5	4	5	4

Table 2. Alignment table. ℓ: left, t: top, c: diagonal.

		a	a	b	c	a	b	c	d
		ℓ	ℓ	ℓ	ℓ	ℓ	ℓ	ℓ	ℓ
a	t	c	ℓ	ℓ	ℓ	ℓ	ℓ	ℓ	ℓ
b	t	t	t	c	ℓ	ℓ	ℓ	ℓ	ℓ
a	t	t	c	t	t	c	ℓ	ℓ	ℓ
c	t	t	t	t	c	t	t	c	ℓ
b	t	t	t	c	t	t	c	t	t
d	t	t	t	t	t	t	t	t	c

(the last row and last column) gives the alignment of the two strings. Table 2 presents the table that is used to calculate the optimal alignment for the pair of strings. One possible alignment for the strings is as follows: `a-b-a-cbd` and `aabcabc-d`. We refer the reader to [5] for a more elaborate discussion.

Our comparison module operates over a pair of traces generated by instrumenting the binaries to be compared as they execute on the same input. To provide an analogy, if the trace is viewed as a string, the equivalence of an alphabet in the string here is a tuple <*Memory Location*, *Operation*, *Value*>. We use the dynamic programming technique to compute an alignment between the pair of traces. The optimality of an alignment is dependent on the cost function used which can be defined in many ways. In this paper, we consider a simple notion of optimality. Gaps in an alignment have unit cost, while all other alphabets have zero cost. Thus an optimal alignment is one that has the smallest number of gaps; observe that for any pair of strings, there maybe many such optimal alignments.

Given memory traces of length m and n for two versions, the time complexity of dynamic programming is $O(mn)$. Thus, even traces of modest length (approximately 15K) can considerably slow down the comparison process. Indeed, for some applications, there are a several million reads or write operations to memory. To make our approach scalable, we employ a heuristic that performs dynamic programming piece-wise on smaller substrings. The heuristic is based on the observation that there is likely to be sufficient locality to apply dynamic programming on the strings yielded by subtraces to yield a good, if not necessarily optimal, alignment. Our heuristic works as follows:

1. Obtain a prefix of fixed length r from both traces.
2. Apply dynamic programming on the prefixes obtained.
3. Find the farthest location in each prefix respectively after which there is no alignment between the prefixes.
4. Obtain a prefix of r starting from these locations respectively from each trace and repeat the process from Step 2.

We use the example discussed above to explain the heuristic. Fix r to be three. In the first step, prefixes `aba` and `aab` are extracted. Aligning these prefixes,

we get `-aba` and `aab-`. In the next step, we extract `acb` from the first string and `cab` from the second string. Aligning the prefixes, we get `ac-b` and `-cab`. Subsequently, we extract `d` and `cd` and align them as `-d` and `cd` respectively. The final alignment is `-abac-b-d` and `aab-cabcd`.

3.3 Extract from Bzip2

We present an extract from `bzip2` to show that ignoring memory locations can indeed lead to reduced precision in detecting variations across program versions. The following piece of code is extracted from the function `generateMTFValues` in `bzip2`, version 0.9.5d.

```
163 void generateMTFValues ( EState* s )
213       ll_i = s->unseqToSeq[block[j] >> 8];
225             j++;
233             zPend--;
240                   s->mtfFreq[BZ_RUNA]++;
247          mtfv[wr] = j+1; wr++; s->mtfFreq[j+1]++;
```

The following piece of code is extracted from the same function in bzip2, version 1.0.2

```
164 void generateMTFValues ( EState* s )
211       ll_i = s->unseqToSeq[block[j]];
219             zPend--;
226                   s->mtfFreq[BZ_RUNA]++;
250             mtfv[wr] = j+1; wr++; s->mtfFreq[j+1]++;
```

In the *learning* process for aliasing, it was observed that memory location of variable in `line 225` in the older version is associated with `lines 211,219, 226, 250`. However none of the lines have a high log-odds ratio with `line 225`. All of them have higher log-odds ratio with some other line in the older version (`lines 213, 233, 240, 247,` respectively). By examining the source code of both the versions, it is obvious that the loop associated with variable `j` has been rewritten and a similar construct is not available in the newer version. By ignoring memory locations, `line 225` was aligning with other unrelated lines, leading to imprecise alignments and therefore reduced precision in the impact analysis. As our experimental results show, memory aliasing can reduce the number of realignments that are observed in affected functions.

4 Evaluation

4.1 Experimental Setup

We provide a comparison of detecting variations across program versions with and without memory aliasing using two versions of the following software packages: `bzip2` [4], `bunzip2` [4], `gawk` [8], `htmldoc` [13] and `wget` [18]. All of these programs are written in C. The details on the versions used for the benchmarks,

the lines of code, the number of functions and other parameters are given in Table 3. We explain the significance of the other columns below. The test cases used for the benchmarks are either randomly generated or are from standard test suites available for them.

Table 3. Benchmark Information and Results (Time in seconds)

	Old Version	New Version	LoC (in K)	Total Fns.	Longest Trace (10^3)	Total Tests	Instr. Time	Analysis Time		Memory (in MB)	affected %
								No Alias	Alias		
`bzip2`	0.9.5d	1.0.2	9	107	6099	107	2631	991	3121	351	25.4
`bunzip2`	0.9.5d	1.0.2	9	107	1839	101	1297	351	2637	89	26.6
`gawk`	3.1.3	3.1.4	41	522	3598	133	1390	450	368	670	41.7
`htmldoc`	1.8.23	1.8.24	65	246	1399	101	3451	970	996	84	48.4
`wget`	1.6	1.7	28	313	158	105	1025	263	232	16	44.4

We perform our tests on Linux 2.6.11.10 (Gentoo release 3.3.4-r1) system running on a Intel(R) Pentium(R) 4 CPU 3.00GHz with 1GB memory. The version of the PIN [16] tool used was a special release 1819 (2005-04-15) for Gentoo Linux. The sources were compiled using GCC version 3.3.4.

4.2 Results

In our current implementation, a list of functions that need to be instrumented and the pair of functions to be considered for comparison are also provided. The number of memory reads and writes, the associated values yielded, the memory locations used, and the line in the source responsible for such an action is presented as output of the instrumented program executed under PIN. By performing this process for both versions, we have two traces of heap reads and writes, and corresponding information that is provided as input to the comparison module. The instrumentation time given in Table 3 includes the time taken to insert instrumentation code and time taken to execute the instrumented version of the binary for all the test cases.

When no aliasing is employed, the operation values and types are compared and aligned. We use a block size of 50 based on the heuristic given in the previous section for a piece-wise alignment of the traces. Furthermore, related memory locations are aligned and the log-odds ratio computed as mentioned earlier in the paper. The comprehensive log-odds ratio is computed by totalling the individual ratios. It is appropriate to note here that memory locations in any two different runs of the same program may be different. Therefore, to ensure consistency, we associate the memory location with the line number in the source. Notice that this approximation may lead to a loss of precision in the presence of two or more memory locations in the same line. Determining a better technique to ensure consistency across test runs is part of our ongoing research. The idea behind the current approximation of memory locations to line numbers is that even though the locations may be different across two runs of the same program, the line numbers are still consistent. More importantly, such an approach can

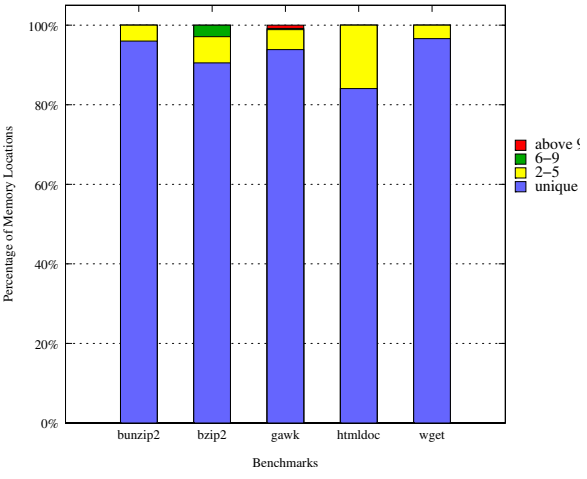


Fig. 4. The percentage of memory locations and the mappings associated

be deployed across program versions as the line numbers need not correlate. On performing the entire process as mentioned above for each test case, we obtain the regions (in the form of line numbers) in the newer version that differ from the older version.

In Figure 4, we present a histogram, which shows the distribution of the memory locations with respect to the number of mappings for each of the benchmarks. We expect that the majority of the locations have unique correspondence, since generally newer versions do not deviate widely from the existing version. Our intuitions are confirmed by the results shown in the figure. Most memory locations (approximately 90%) in the older version have unique correspondence (perfect mapping) with memory locations in the newer version. The remaining locations have multiple mappings and these are the locations that can reduce the effectiveness of impact analysis. We observe that among these memory locations, most of them correspond to two or three other memory locations in the newer version. In the case of `gawk` [8], we observe as many as 20 different locations in the newer version that can be mapped to one memory location. However such occurrences are rare. Such increased correspondence signifies that the memory location cannot be aliased with any other memory location and the observed alignments are accidental. Our extract from `bzip2` [4] in the previous section corroborates this observation.

When aliasing is used in our comparison tool, we ensure that the memory locations associated with each memory operation are compared and two tuples are aligned if and only if the operation value and type are equal and the associated memory locations are aliases. In Figure 5, we present the results of the improvement using this approach. The figure shows the distribution of functions with respect to the change (in percentage) of the number of lines within an impacted function when aliasing is used, as opposed to no aliasing. We observe that there is a significant change in the number of lines affected, even though only a few

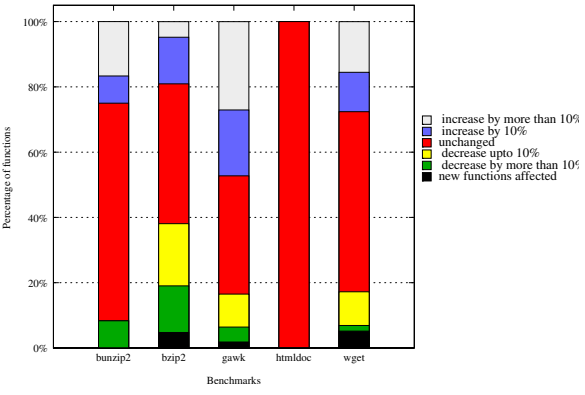


Fig. 5. Change in size of impacted regions when aliasing detection is used

memory locations had multiple mappings (from Figure 4). Approximately 50-60% of the functions do not change as a result of using aliasing. In some cases, for example in `htmldoc` [13], we observe that even though the uniquely mapped memory locations is the least over all the benchmarks, there is, nonetheless, no change in the affected regions.

In Table 3, we provide the specifics of our benchmarks and the results obtained using our technique. The number of lines of code varies from 9K to 65K with the number of functions varying from 100 to 500, approximately. The length of the trace represents the number of reads and writes to the heap in thousands of instructions. The longest trace observed is approximately 6 million for `bzip2`. The average memory used, while significant, is not problematic. This is expected for many dynamic analysis scenarios because precise information on heap operations is being gathered. No significant difference in memory utilization was observed when aliasing was used and is therefore not shown as a separate column in the table. The percentage of affected regions is also provided in the table. The affected percentage reveals that a sizeable fraction of the newer version of a benchmark program is impacted by changes to the older.

The time taken for our technique is composed of the instrumentation time of the binary and execution time of comparison module. There are two reasons for the inefficiency of the instrumentation process. The first is the use of a dynamic binary instrumentation tool as opposed to static instrumentation. Therefore for each test case, we insert appropriate instrumentation code. We believe the time taken for this can be significantly reduced using alternative instrumentation strategies. The current impact analysis tool currently tracks all heap related operations. This can also play an important role in increasing the instrumentation time. The length of the trace and instrumentation time are directly correlated. For example, `wget` has a shorter trace and thus significantly smaller instrumentation time compared to `bzip2`. One way to reduce the number of heap operations tracked is to discard operations found in regions already known to have been

affected from previous test runs. The difference in the analysis time of the two approaches is not significant except for `bzip2` and `bunzip2`, where we observe that the approach using aliasing requires more time. Note that in the aliasing approach, since memory locations are compared for every alphabet mapping, the increased time can be attributed to the number of memory locations occurring in the benchmark. Since we expect that our approach will be generally used offline, we believe the improved precision resulting from taking aliasing information into account outweighs the added costs for performing location comparisons.

5 Related Work

Our approach is motivated by similar techniques employed for problems in bioinformatics. PAM [6] and BLOSSOM [9] are two commonly used substitution matrices for detecting amino acid substitution. Amino acids can mutate with time and it is necessary that two different amino acids be aliased to give a better sequence alignment. Analogously, we have multiple versions (the newer version can be considered as the mutation of the older version) and to align these newer versions, it is necessary that the memory locations in the two versions be aliased. We address this problem in the paper and use the log-odds ratio to alias memory locations. We also show by way of experiments on many open source systems that the memory aliasing approach employed here enhances the precision of the impact analysis.

Zhang and Gupta [19] present a novel method for matching dynamic execution histories across program versions for detecting bugs and pirated software. They use an offset-based aliasing technique to correlate stack locations across the two programs. While this is applicable in many cases, it is not clear whether this approach is generalizable. In this paper, we abstract the problem of memory aliasing into one of finding longest common subsequence of two strings composed of different alphabets. We also present a probabilistic solution to this problem.

There has been much prior work in impact analysis for improving testing efficiency in the presence of program changes [1, 15, 17]. In these approaches, functions that follow a modified function in some execution path are added to the impact set. We share obvious similarities with these efforts, but differ both in the mechanisms used to identify impacted functions, and the ability to identify localized regions of change within these functions. Furthermore, since our technique operates over binary executables, we are not reliant on program analysis of input sources or programmer annotations.

Our approach can also be extended to correlate variables across program versions to check whether the invariants are preserved across these versions. Ernst *et. al.* provide a technique for automatically detecting invariants within a single program version [7]. However, it is not clear how invariants across program versions can be correlated, in the presence of variable renaming, deletions, additions, and general changes in the program's data- and control-flow . By proposing a

new formulation for generating the initial history of alignments and applying the log-odds ratio, we believe variables can be correlated even under these circumstances.

Pointer analysis is a well studied problem and a number of techniques in the literature is discussed by Hind and Pioli [11]. "A pointer alias analysis attempts to determine when two pointer expressions refer to the same storage location" [10]. However, many of the techniques discussed in the literature is for alias analysis within a single program. In this paper, we address an entirely different problem of 'must' aliasing [14] of pointers across two program versions. The technique presented in the paper provides a solution based on statistical correlation.

Dynamic programming, more specifically longest common subsequence (LCS), is used in many applications. One such application in software engineering is described in [3]. The foundation of their approach that for similar bugs, the call stack also shares similarities. Therefore, by pruning unnecessary information from the call stack, and comparing the resulting string representation with an existing signature, a score can be computed to the match using a longest common subsequence algorithm. The similarity between their approach and ours is restricted to the underlying technique and its applicability in a software engineering context, but does not extend to impact analysis or variation detection across program revisions.

6 Conclusions

This paper describes a technique for identifying memory aliases across program versions. Our approach works by collecting traces of program executions, in which each element of the trace contains an operation, value, and a memory location; trace results applied to the same test input on the versions being compared are aligned without considering the memory locations used by the program. By computing the log-odds ratio between memory locations based on the above alignments on a few test inputs, we obtain an aliasing of the locations across versions. To validate our approach, we compare program versions based on traces with and without aliasing detection on a number of open-source programs. Experimental results show that our technique improves the precision of identifying impacted regions significantly. Our approach can be easily extended to other applications where memory aliasing is required. For example, as part of ongoing work, we are investigating the use of this approach for correlating variables across program versions to test preservation of invariants and program matching. Another avenue for future work is to evaluate the applicability of our approach to more heap-centric languages such as Java.

Acknowledgements. This work is supported by National Science Foundation Grants CCF-0444285, CNS-0334141 and STI 501-1398-1078. We also like to thank the anonymous reviewers for their constructive feedback.

References

[1] T. Apiwattanapong, A. Orso, and M. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 432–441, 2005.

[2] http://www.ncbi.nlm.nih.gov/education/ blastinfo/information3.html.

[3] M. Brodie, S. Ma, G. Lohman, T. Syeda-Mahmood, L. Mignet, N. Modani, M. Wilding, J. Champlin, and P. Sohn. An architecture for quickly detecting known software problems. In *ICAC 2005: Proceedings of the International Conference on Autonomic Computing*, 2005.

[4] http://www.bzip.org.

[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 2nd edition, 2001.

[6] M.O. Dayhoff, R.M. Schwartz, and B.C.Orcutt. A model of evolutionary change in proteins. matrices for detecting distant relationships. *M. O. Dayhoff, (ed.), Atlas of protein sequence and structure, volume 5, pp. 345–358 National biomedical research foundation Washington DC.*, 1979.

[7] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001.

[8] http://www.gnu.org/software/gawk/gawk.html.

[9] S. Henikoff and J. Henikoff. Amino acid substitution matrices from protein blocks. In *Proc. National Academy of Sciences, USA*, 1992.

[10] M. Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.

[11] M. Hind and A. Pioli. Which pointer analysis should i use? In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[12] D. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of ACM, 24(4)*, pages 664–675, 1977.

[13] http://www.htmldoc.org/.

[14] S. Jagannathan, P. Thiemann, S. Weeks, and A.K. Wright. Single and loving it: Must-alias analysis for higher-order languages. In *Symposium on Principles of Programming Languages*, pages 329–341, 1998.

[15] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 308–318, 2003.

[16] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[17] A. Orso, T. Apiwattanapong, and M. Harrold. Leveraging field data for impact analysis and regression testing. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 128–137, 2003.

[18] http://www.gnu.org/software/wget/.

[19] X. Zhang and R. Gupta. Matching execution histories of program versions. In *Proceedings of the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE)*, pages 197–206, Sep, 2005.

The Pervasiveness of Global Data in Evolving Software Systems*

Fraser P. Ruffell and Jason W.A. Selby

School of Computer Science, University of Waterloo,
Ontario, Canada, N2L 3G1
{fruffell, j2selby}@uwaterloo.ca

Abstract. In this research, we investigate the role of common coupling in evolving software systems. It can be argued that most software developers understand that the use of global data has many harmful side-effects, and thus should be avoided. We are therefore interested in the answer to the following question: if global data does exist within a software project, how does global data usage evolve over a software project's lifetime? Perhaps the constant refactoring and perfective maintenance eliminates global data usage, or conversely, perhaps the constant addition of features and rapid development introduce an increasing reliance on global data? We are also interested in identifying if global data usage patterns are useful as a software metric that is indicative of an interesting or significant event in the software's lifetime.

The focus of this research is twofold: first to develop an effective and automatic technique for studying global data usage over the lifetime of large software systems and secondly, to leverage this technique in a case-study of global data usage for several large and evolving software systems in an effort to reach answers to these questions.

1 Introduction

A focus of the software engineering discipline has been, and continues to be, the development and deployment of techniques for yielding reusable, extensible, and reliable software [3]. One proven approach toward obtaining these goals and others, is to develop software as a collection of independent modules [9, 6]. This technique is especially effective when the individual modules experience a low-degree of inter-dependence or *coupling* [18, 11]. Modules which are self-contained and communicate with others strictly through well-defined interfaces are not likely to be affected by changes made to the internals of other unrelated components.

Although designing software which exhibits a low degree of coupling is highly desirable, if the modules of a software system are to communicate at all, some form of coupling must exist. In [11] the following seven types of coupling are

* This research was supported in part by a Natural Sciences and Engineering Research Council of Canada Strategic grant.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 396–410, 2006.

defined in increasing severeness: no coupling, data coupling, stamp coupling, control coupling, external coupling, common coupling, and content coupling.

The focus of this paper is on the second most undesirable form: common coupling. This manifestation of coupling implicitly occurs between all modules accessing the same global data. In [20], this form of coupling is referred to as *clandestine* coupling. Many different programming languages, old and new alike, provide support for global data, and as illustrated later in this paper, common coupling is rampant in many large software systems.

In this research, we investigate the role of global data in *evolving software systems*. It can be argued that most software developers understand that the use of global data has many harmful side-effects and thus should be avoided. We are therefore interested in the answer to the following question: if global data *does* exist within a software project, how does global data usage *evolve* over a software project's lifetime? Perhaps the constant refactoring and perfective maintenance eliminates global data usage or, conversely, perhaps the constant addition of features and rapid development introduce an increasing reliance on global data? We are also interested if global data usage patterns are useful as a software metric. For example, if a large number of global variables are added across two successive versions, is this indicative of an interesting or significant event in the software's lifetime? The focus of this research is twofold: first to develop an effective and automatic technique for studying global data usage over the lifetime of large software systems and secondly, to leverage this technique in a case-study of global data usage for several large and evolving software systems in an effort to attain answers to these questions.

This paper is organized as follows. Section 2 discusses the many pitfalls and possible reasons for using global data. Section 3 then presents our thoughts and expectations on global data usage in the three software systems examined. In Section 4, an in-depth discussion on our tool `gv-finder` is presented. Section 5 provides an overview of the systems examined in this study, followed by the results and analysis of applying `gv-finder` to these systems in Section 5. Finally, Section 7 concludes.

2 Global Data

The notion of *scope* is intrinsic to the declaration of source code entities (e.g. class, function, datum) in all programming languages. Scope is simply a maximal region of code for which the declared source code item is bound to [6]. Depending on language semantics, the scope may vary from a single compound statement (local scope), to any and every source file in a software project (global scope). The focus of this paper is on the set of variables within an application's source code that are declared with a global scope. As discussed later in Section 4 we further build upon this definition to include other variables of interest.

Global variables can be used indiscriminately in any module within a software project, and thereby all modules referencing the same global data are implicitly coupled. For this reason, it is well known that the use of global variables poses

a real and serious threat to software maintainability. The resulting rampant coupling can greatly impair program comprehension through a wide range of unanticipated side-effects; from hidden aliasing problems, namespace pollution, to even hampering code reuse across projects. Without a full understanding *all* of these and other less obvious implications of using global data, misconceptions about the safety of read-only access to globals, or the judicious use of file scope globals (e.g. statics in `C`) will continue to exist. For an in-depth discussion as to why global variables are harmful, the reader is referred to [6, 9, 17, 20].

Despite the well known drawbacks of using global data, there are a small number of *valid* reasons justifying their use. For example, global data can be used to emulate named constants and enumerations in those languages which do not support them directly [9]. However, situations which require the use of global data are rare and can almost always be avoided.

Previous work by Briand *et al.* [2] found that the degree of common coupling inside of a system is related to fault-proneness. Yu *et al.* [20] and Schach *et al.* [16] also examined common coupling in terms of the effects on maintainability and quality, respectively.

The application of data mining to various entities of the software development process to discover and direct evolution patterns has recently received extensive treatment, most notably in [4, 13, 21].

3 Exploration of Global Variables in Software Evolution

Two opposing views of software evolution exist. In the first view, early releases of a software project are seen as pristine, and that as the software *ages*, entropy takes hold and it enters into a constant state of decay and degradation [12]. Accordingly, one may hypothesize about the pervasiveness of global data with this view in mind:

> *Since evolving software is in a perpetual state of entropy, the degree of maintainability will decrease partially due to an increase in both the number and usage of global variables within an aging software project.*

Conversely, another view is that software is in a constant state of refactoring and redesign and, along with perfective maintenance, one can conclude that the early releases of a software project are somewhat unstructured, and as the project *ages* the design and implementation become more stable and mature. With this view in mind, one can suggest the following about the pervasiveness of global data in evolving software:

> *As software evolves in an iterative development cycle of constant refactoring and redesign, the degree of maintainability will increase partially due to an increase in both the number and usage of global variables within a growing software project.*

Although both hypotheses are convincing when viewed in isolation, it appears to us that it is more likely that neither will apply uniformly to *all* evolving soft-

ware systems. Instead, we propose that the defining characteristics of each software system (such as the development model, development community, relative age, project goals, etc.) are the factors determining which viewpoint is more influential. In particular, we adopt the three-pronged classification of Open Source Software (OSS) as defined in [10]. The three types of OSS, and predictions on the global data usage for each, are:

1. **Exploration-Oriented**. Research software which has the goal of sharing knowledge with the masses. Such software usually consists of a single main branch of development, which is tightly controlled by a single leader. In such a project we predict to see very few global variables and, as the software evolves, a decrease if any change in global variable usage.
2. **Utility-Oriented.** This type of software is feature-rich, and often experiences rapid development, possibly with forks. In this category of software, we expect to see a relatively high reliance on global data, which will gradually increase over the software's lifetime, possibly with periods of refactoring.
3. **Service-Oriented.** Software in this category tends to be very stable and development is relatively inactive due to its large user-base and small developer group. Unlike exploration-oriented software, where a single person has complete authority, a small number of "council" members fulfill the decision-making role. For software of this classification, we predict global data usage to be higher than exploration-oriented software but less than utility-oriented software. As the software evolves, we also expect to observe a decrease in reliance upon global data.

4 Methodology

Our objective was to study both the presence and role of global data in several large-scale software systems, and therefore, it was important to devise an approach for *automatically* collecting such data. Whereas in [20], global data usage was collected for a *single* version of the Linux kernel, our focus was more extensive, as we were interested in examining *numerous* versions of multiple software projects. Consider one of the three case studies presented later in Section 5: *GNU Emacs*. In total, 15 versions of Emacs were examined (across a 14 year time period), the accumulative source code base consists of roughly 4 million lines of code. Clearly, examining the pervasiveness of global data over the evolution of such a large-scale software system requires an automated process. This section provides an overview and discussion of the design and implementation of our global data collection tool called `gv-finder`.

Initial approaches to developing the global data collection tool included hand coding a parser, and the modification of `gcc`. However, the approach decided upon was to write a stand-alone tool similar to a linker, which takes as input a collection of relocatable object files. Relocatable object files are usually produced as the output from either a compiler or assembler, and contain the machine code representation for some source code entity (e.g., a file or a concatenation of files)

along with information needed by both the linker and loader [15]. The following two observations lead us to adopt this approach:

- If a source file uses global data which happens to be instantiated within a different source file, the corresponding relocatable object file will contain a symbol table entry indicating that the global data is undefined. When the linker is invoked with the complete set of object files used for constructing the target application, it will replace any reference to the undefined global data with the address of the global data instantiated in one of the other object files.
- If a source file instantiates and exports global data, the corresponding relocatable object for that file will contain a symbol table entry declaring the data as global. The linker uses the address of a global symbol (also found in the symbol table) to resolve references to the same global data occurring in other external source files, as well as for any internal references.

Therefore, by inspection of the set of object files which constitute the final executable application, one can determine (a) the names of all global data and the corresponding module in which they are defined and (b) for each global data, the name of the modules which refer to it. In addition to satisfying all of our design criteria, this method offers the advantage of being portable across different compiler suites. This may be useful if an application only compiles with a certain version of a compiler, or a specific company's compiler — native compilers for a given platform target a common standard object file format.

Our analysis of relocatable ELF object files makes the following distinction between different types of global data:

- **External Global Data.** If one or more object files contain an undefined reference to global data, but no object file is found to provide a matching definition, we consider the global data to be *external* to the application. This occurs when the application makes use of a library which exports global state. A common example is the use of `stdout` from the `C` standard library. This is the least severe type of global data since the application itself is not responsible for the design of the libraries it depends upon.
- **Static Global Data.** If an object file contains a definition of global data which is marked as "local" then the global data is classified as *static*. This occurs in languages such as `C` and `C++` where global data is declared with the `static` keyword[7, 19]. Static global data can only be used in the file which declares the variable, and therefore can not introduce "clandestine" coupling [20] with other external modules. However, all the other disadvantages associated with using global data are applicable to static data, and therefore we feel it is important to make the distinction as static data is still potentially dangerous and undesirable.
- **True Global Data.** If an object file contains a definition of data marked as "global", the data is then classified as *true global data*. This data can be referenced in any other module without restriction, simply by referring to the data's name. This is the most dangerous type of global data since every

module which references the exported global variable becomes implicitly coupled [20].

It should be noted that this approach to global data analysis requires the target application to be compiled. This turned out to be a challenge for the very early versions of the software studied in Section 6, as language standards, system header files, and the required build tools have also evolved independently, and tend not to be backward compatible. However, for global variable analysis, it is only required that the source files compile, even if the resulting executable does not run correctly (or at all). Therefore, with a relatively small investment in time, we found that many of the older versions could be compiled by strategically adding fix-up macros, re-using configuration files across different versions and, in the worst-case scenario, simply removing offending lines of code (less than 100 lines of code were commented out in any given release).

5 Case Study

Over the course of this study we examined one example of each of the three classifications of OSS projects defined in [10]. Specifically, the Vim, Emacs and PostgreSQL projects were examined.

5.1 Vi IMproved (Vim)

The Vi IMproved (Vim) editor began as an open-source version of the popular VI editor, and has now eclipsed the popularity of the original Vi. Vim was created by Bram Moolenar, who based upon it another editor, Stevie[1]. Development of Vim centres around Moolenar, with other developers contributing mostly small features, however, the process relies upon the user community for bug reports. In terms of the classification of open-source software defined in [10], Vim is considered an example of a *utility-oriented* project.

Sixteen releases of Vim dating back to 1996 were studied (four earlier versions which target the Amiga were unanalyzable). Table 1 displays the Vim chronology of the examined releases. Most of the releases are considered minor, however, releases 5.3 and 6.0 are major, contributing at least 50 KLOC each to the system.

Table 1. Chronological data for the releases of Vim examined in this study [5, 8]

Release	Date	SLOC	Total LOC	Release	Date	SLOC	Total LOC
4.0	05/1996	43594	59966	5.5	09/1999	94247	127055
4.1	06/1996	43891	60396	5.6	01/2000	94964	128102
4.2	07/1996	44017	60600	5.7	06/2000	96225	129681
4.3	08/1996	44621	61606	5.8	05/2001	95548	128864
4.4	09/1996	44693	61751	6.0	09/2001	140182	187196
4.5	10/1996	44742	61875	6.1	03/2002	142091	189632
5.3	08/1998	79260	107876	6.2	06/2003	156700	209680
5.4	07/1999	93771	126383	6.3	06/2004	162441	217501

5.2 GNU Emacs

The Emacs editor is one of the most widely used projects developed by GNU. It was originally developed by Richard Stallman, who still remains the project maintainer. Given the development process and community that supports Emacs, we consider it to be an *exploration-oriented* project.

Our examination of Emacs consisted of fifteen releases stretching as far back as 1992. Details pertaining to the releases that we studied can be found in Table 2.

Table 2. Chronological data for the releases of Emacs examined in this study

Release	Date	SLOC	Total LOC	Release	Date	SLOC	Total LOC
18.59	10/1992	56216	74752	20.5	12/1999	105324	146655
19.25	05/1994	75412	104608	20.6	02/2000	105336	146693
19.30	11/1995	80824	112780	20.7	06/2000	105437	146849
19.34	08/1996	100514	140000	21.1	10/2001	137615	197481
20.1	09/1997	100406	140357	21.2	03/2002	137835	197814
20.2	09/1997	100408	140357	21.3	03/2003	138035	198130
20.3	08/1998	104193	145258	21.4	02/2005	138035	198130
20.4	07/1999	105170	146422				

5.3 PostgreSQL

As an example of *service-oriented* OSS, the PostgreSQL relational database system was examined. PostgreSQL is an example of a exploration-oriented (research) project that has morphed into a service-oriented project. The system was initially developed under the name POSTGRES at the University of California at Berkeley[14]. It was soon released to the public and is now under the control of the PostgreSQL Global Development Group.

Table 3. Chronological data for the releases of PostgreSQL examined in this study

Release	Date	SLOC	Total LOC	Release	Date	SLOC	Total LOC
1.02	08/1996	102965	175538	7.4	11/2003	222694	349461
6	07/1997	98062	162253	8.0.0	19/01/2005	242887	382686
7.2	02/2202	252155	276496	8.0.1	31/01/2005	242991	382865
7.3	11/2002	194822	308305				

We studied seven releases, of which three are considered major releases (1.02, 6.0 and 8.0.0). Version 1.02 (aka Postgres95) was the first version released outside of Berkeley, and incorporated a SQL frontend into the system. Although, the PostgreSQL project is composed of many programs, we limited our study to the PostgreSQL backend server. Table 3 outlines the date and size changes of the PostgreSQL server for the releases examined.

6 Experimental Results and Discussion

In this section we report and discuss the results gathered through the use of `gv-finder` on the selected open-source projects. Specifically, we examine the evolution of the projects in terms of their size (lines of code), the number of global variables referenced, their reliance upon global variables, and finally, the extent to which global data is used throughout the system.

6.1 Changes in Number of Lines of Code

Over the lifetimes of the projects that we studied, each has at least doubled in terms of their code size. Size data collected includes uncommented, non-white space source lines of code (SLOC), and total lines of code (LOC). Referring back to Tables 1, 2, and 3 we see the changes in source lines of code as well as total lines of code for Vim, Emacs and PostgreSQL, respectively.

As expected, each project shows a small increase in size over the minor releases as a result of perfective maintenance which can be attributed primarily to bug fixes. However, the large increases stems from the the major releases when new features were added to the systems.

Interestingly, the LOC decreases substantially from version 7.2 to 7.3 of PostgreSQL. Examination of the documented changes revealed that support for a specific protocol was removed. However, it is unclear if this change alone accounts for the 70 KLOC that was removed from the system.

6.2 Evolution of the Number of Global Variables

Initially it was hypothesized that the number of global variables would decrease over the lifetime of a project as the developers had more time to perform corrective maintenance and replace them with safer alternatives. However, this was not what was discovered. In fact, we found that the number of global variables present in *all* of the systems examined grew alongside the code size as demonstrated in Figs. 1, 2, and 3. In the figures, the number of distinct global variables is classified as being either true, static or external. To further clarify the figures, consider Fig. 1. Examination of Vim release 5.3 shows that the total number of global variables identified is 684. These 684 references are composed of 426 true, 238 static, and 20 external global variables.

The finding that the number of global variables increases alongside the lines of code might suggest that the use of global variables is inherent in programming large software systems (at least those programmed in `C`). This is even more interesting given that according to the classifications in [10], PostgreSQL and Emacs are developed under a stringent process that ideally would attempt to limit the introduction and use of global variables.

6.3 Evolution of the Reliance Upon Global Variables

In an attempt to evaluate how reliant the systems are upon global data, we recorded the number of lines of code that reference global data. Using this, we

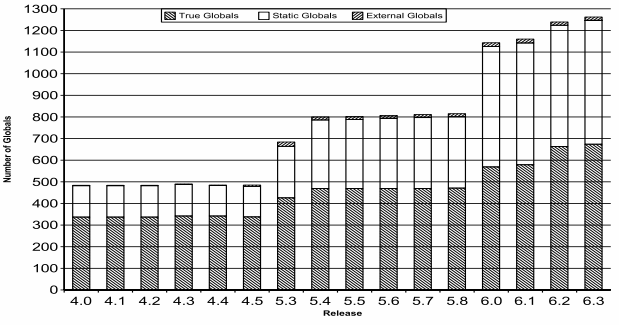


Fig. 1. The number of true, static and external globals identified by `gv-finder` in Vim

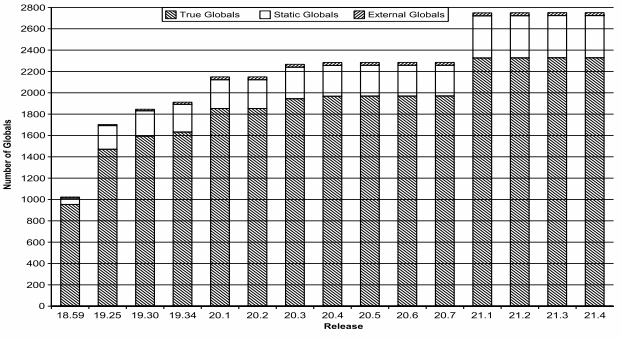


Fig. 2. The number of true, static and external globals identified by `gv-finder` in Emacs

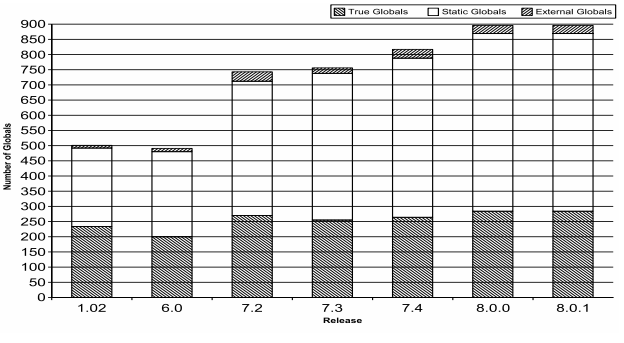


Fig. 3. The number of true, static and external globals identified by `gv-finder` in PostgreSQL

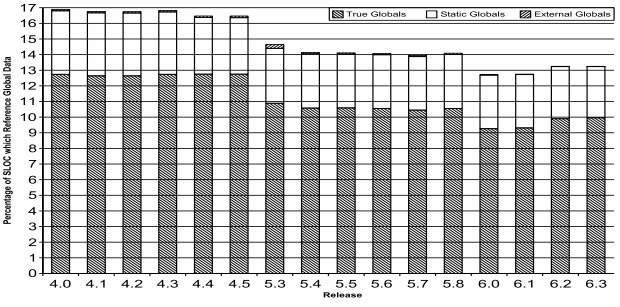


Fig. 4. The percentage of references to global data per line of source code. The percentages are classified as either true, static and external globals as identified by `gv-finder` in Vim.

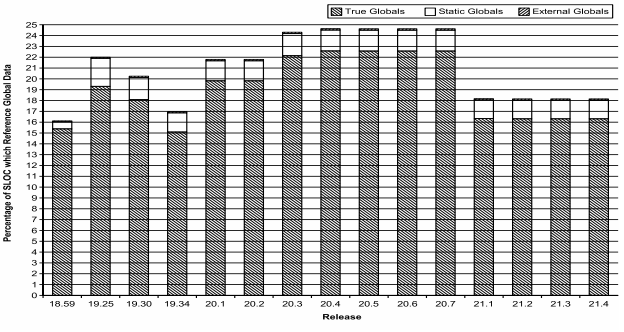


Fig. 5. The percentage of references to global data per line of source code identified by `gv-finder` in Emacs

are able to report the percentage of lines of source code (SLOC) that reference global data as displayed in Figs. 4, 5, and 6.

In this form the figures diminish the actual reliance of the projects upon global data. This can be attributed to two factors. First, each line that references a global variable is only counted once, even if it might reference multiple global variables. However, the most important factor is simply that the number of lines of code is growing much faster than the number of globals. Therefore, even though the number of global variables present in each system is growing, the use of SLOC as the divisor negates this fact.

To gain a better perspective on the reliance of global data, we plotted the number of references to global data divided by the total number of globals. Examining Figs. 7, 8, and 9, a wave pattern is observed for all projects. This might indicate that the original intuition that global variables were added to code as a quick fix in order to ship the initial release, after which their number would decrease, was simply too limited. The wave pattern that is evident in the figures could be interpreted as the iterative process of adding new features, and hence new globals, to the system and the later factoring out of them over time.

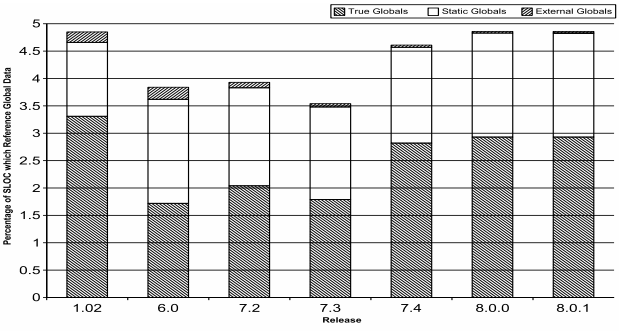


Fig. 6. The percentage of references to global data per line of source code identified by `gv-finder` in PostgreSQL

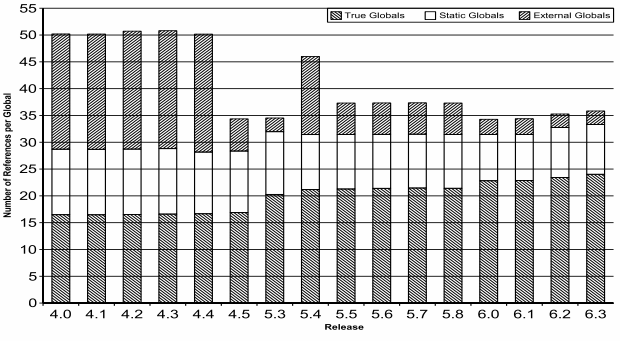


Fig. 7. The number of references per global variable discovered by `gv-finder` in Vim

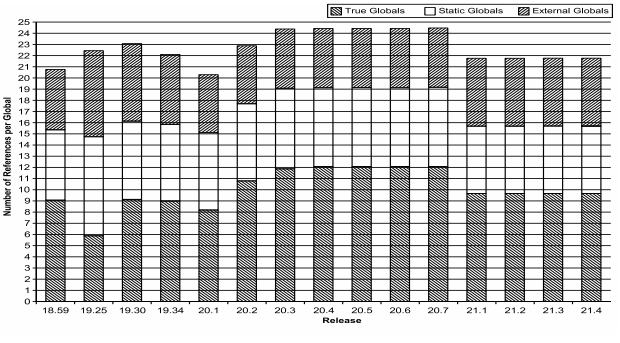


Fig. 8. The number of references per global variable discovered in Emacs

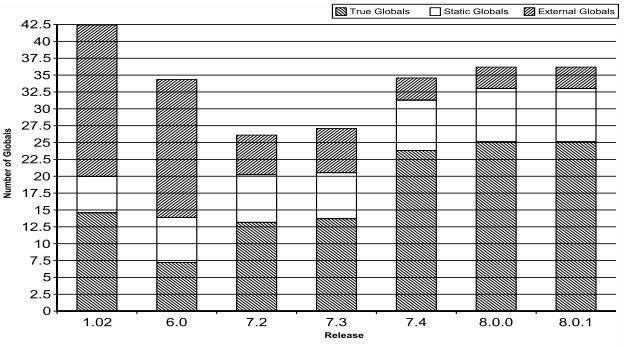


Fig. 9. The number of references per global variable discovered in PostgreSQL.

However, contrary to our intuition the reliance upon global data appears to peak at mid-releases. This may indicate that the addition of new features in major-releases are the result of clean, well-planned designs. It appears that the process of identifying bugs and patching them as quickly as possible results in the introduction of the majority of references to global data. As the frequency of bug reports curtail the developers are able to focus on refactoring the hastily coded bug fixes, thereby reducing the reliance upon globals.

6.4 Evolution of the Extent of Use of Global Variables

Finally, in order to examine how widespread the use of global variables is throughout the systems, we collected data pertaining to the number of functions which make use of global data as displayed in Figs. 10, 11, and 12. The extent of usage of global data in Vim and Emacs is considerably higher than in PostgreSQL. The percentage of functions which reference global data is greater than 80% for both of the editors, while the percentage in PostgreSQL is approximately 45%.

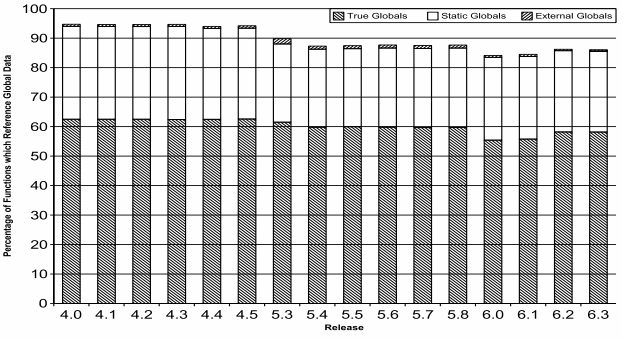


Fig. 10. The percentage of functions which reference global data. The percentages are classified as either true, static and external globals as identified by `gv-finder` in Vim.

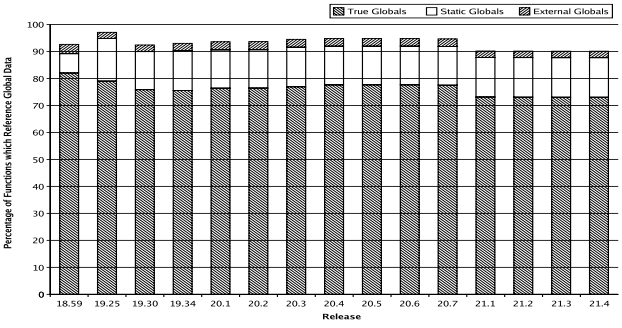


Fig. 11. The percentage of functions which reference global data as identified by `gv-finder` in Emacs

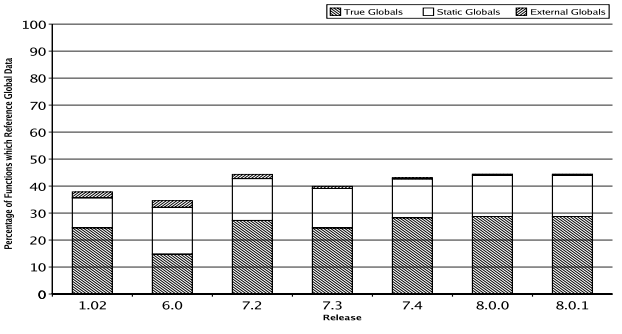


Fig. 12. The percentage of functions which reference global data as identified by `gv-finder` in PostgreSQL

We should note that there are some threats to the validity of our work. As noted earlier, we were unable to examine every single release of all three projects. The application of `gv-finder` to all releases would result in a more precise view of the evolution of global data usage across the entire lifetime of the projects. However, we believe that the examined releases provide sufficient insight into the projects in order to base our findings.

Additionally, the usage pattern of global data discovered by our work may not be visible in other types of software. Specifically, our findings are the result of the examination of open-source projects, two of which are text editors. Therefore, it is not clear if our results would hold for a wider spectrum of software (for example, closed-source projects). In order to draw any further conclusions we plan on examining a larger number of projects, ranging from compilers to multimedia players.

7 Conclusion

In this study we performed a detailed analysis of the pervasiveness of global data in three open-source projects. Our contributions are twofold. First, the categorization of a project as either service-, utility- or exploration-oriented does not appear to be indicative of the usage of global data over its lifetime. In conjunction with the fact that the number of global variables increase alongside the lines of code could indicate that the use of global data is inherent in programming large software systems and can not be entirely avoided. Second, and most interesting is the finding that the usage of global data followed a wave pattern which peaked at mid-releases for all of the systems examined. This might suggest that the addition of new features in major-releases are the result of proper software design principles while the corrective maintenance performed immediately after a major-release may result in increasing the reliance upon global data. Later phases of refactoring (preventative maintenance) appear to be able to slightly reduce this reliance.

Acknowledgments

We would like to Mark Giesbrecht, Michael Godfrey and the students of Prof. Godfrey's CS846 class at UW where this work was initially developed, Cory Kasper, and the FASE referees for their comments on this work.

References

1. Vim F.A.Q. Available at http://vimdoc.sourceforge.net/vimfaq.html.
2. L. C. Briand, J. Daly, V. Porter, and J. Wüst. A comprehensive empirical validation of design measures for object-oriented systems. In *METRICS '98: Proceedings of the 5th International Symposium on Software Metrics*, page 246, Washington, DC, USA, 1998. IEEE Computer Society.
3. F. P. Brooks Jr. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.
4. M. Fischer, J. Oberleitner, J. Ratzinger, and H. Gall. Mininig evolution data of a product family. In *MSR'05: Proceedings of the International Workshop on Mining Software Repositories*, May 2005.
5. S. Guckes. Vim history - release dates of user versions and developer versions. Available at http://www.vmunix.com/vim/hist.html.
6. A. Hunt and D. Thomas. *The pragmatic programmer: from journeyman to master.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
7. B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
8. J. Magid. Historic linux archive. Available at http://www.ibiblio.org/pub/historic-linux/ftp-archives/sunsite.unc.edu/Sep-29-1996/apps/editors/vi/.
9. S. McConnell. *Code complete: a practical handbook of software construction.* Microsoft Press, Redmond, WA, USA, second edition, 2004.

10. K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye. Evolution patterns of open-source software systems and communities. In *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*, pages 76–85. ACM Press, 2002.
11. A. J. Offutt, M. J. Harrold, and P. Kolte. A software metric system for module coupling. *J. Syst. Softw.*, 20(3):295–308, 1993.
12. D. L. Parnas. Software aging. In *ICSE '94: Proceedings of the 16th International Conference on Software Engineering*, pages 279–287. IEEE Computer Society Press, 1994.
13. M. Pinzger, M. Fischer, and H. Gall. Towards an integrated view on architecture and its evolution. *Electronic Notes in Theoretical Computer Science*, 127(3):183–196, April 2005.
14. PostgreSQL Global Development Group. *PostgreSQL 8.0.0 Documentation*, 2005.
15. L. Presser and J. R. White. Linkers and loaders. *ACM Comput. Surv.*, 4(3):149–167, 1972.
16. S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller, and J. Offutt. Quality impacts of clandestine common coupling. *Software Quality Control*, 11(3):211–218, 2003.
17. S. R. Schach and A. J. Offutt. On the nonmaintainability of open-source software position paper. *2nd Workshop on Open Source Software Engineering*, May 2002.
18. W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems J.*, 13(2):115–139, 1974.
19. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
20. L. Yu, S. R. Schach, K. Chen, and J. Offutt. Categorization of common coupling and its application to the maintainability of the linux kernel. *IEEE Trans. Software Eng.*, 30(10):694–706, 2004.
21. T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.

Relation of Code Clones and Change Couplings

Reto Geiger, Beat Fluri, Harald C. Gall, and Martin Pinzger

s.e.a.l. – software evolution and architecture lab,
Department of Informatics,
University of Zurich, Switzerland
{geiger, fluri, gall, pinzger}@ifi.unizh.ch

Abstract. Code clones have long been recognized as bad smells in software systems and are considered to cause maintenance problems during evolution. It is broadly assumed that the more clones two files share, the more often they have to be changed together. This relation between clones and change couplings has been postulated but neither demonstrated nor quantified yet. However, given such a relation it would simplify the identification of restructuring candidates and reduce change couplings. In this paper, we examine this relation and discuss if a correlation between code clones and change couplings can be verified. For that, we propose a framework to examine code clones and relate them to change couplings taken from release history analysis. We validated our framework with the open source project Mozilla and the results of the validation show that although the relation is statistically unverifiable it derives a reasonable amount of cases where the relation exists. Therefore, to discover clone candidates for restructuring we additionally propose a set of metrics and a visualization technique. This allows one to spot where a correlation between cloning and change coupling exists and, as a result, which files should be restructured to ease further evolution.

1 Introduction

Code duplication is often cited as one of the major *bad smells* in software systems [1]. Systems containing a large proportion of duplicated code are considered to be difficult to maintain. It is estimated that normal industrial source code contains 5 – 20 % of duplicated fragments [2]. The financial impact of maintenance is grave – the costs of changes carried out after delivery are estimated at 40 – 70 % of the total costs during a system's lifetime [3].

As bad smells are indicators for maintainability problems, they lose their significance if the system remains stable and is never changed after its initial release. According to Lehman's *Laws of Software Evolution* [4], software systems which are actively used to solve problems in the real world are never completely stable during their lifetime. Basically, a system has to evolve so that its users remain satisfied. In this case the possible negative influence of code clones on the maintainability comes into play. Code duplication increases the change effort and reduces the understandability of the code drastically. Thus, code clones are a major factor that have to be considered during the evolution of a system.

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 411–425, 2006.

A sign of maintainability problems during the evolution of a system are change couplings. Gall *et al.* defined change couplings as files which are committed at the same time, by the same author, and with the same modification description [5]. If such couplings occur only sporadically, they do not present a major problem. If, on the other hand, files are frequently changed together during the evolution of the software system, a refactoring or even reengineering should be considered.

Based on the assumption that whenever a duplicated code fragment is changed its variants also have to change [6], there seems to be a strong relation between code clones and change couplings. In this paper, we investigate whether this relation holds. We present a framework to determine the relation of code clones and change couplings and introduce a visualization technique aiding developers to choose which code clones to refactor. We further present a validation of our framework with the large open source project Mozilla. The results of the validation show that although the relation is statistically unverifiable it derives a reasonable amount of results where the relation exists. Furthermore, it shows that based on the relation data our visualization technique can be used to identify the candidates for a refactoring.

The remainder of the paper is organized as follows: Section 2 presents related work that has been done in the area of code clone detection and the impact of these duplications to the evolution of software systems. In Section 3 we describe our framework that has been applied to the case study. Section 4 presents a validation of our framework and discusses its results. We conclude the paper in Section 5 and indicate areas for future research.

2 Related Work

A large number of code clone detection techniques have been developed. Four different general approaches can be discerned: detection based on lexical analysis [7, 8, 9], on source code metrics [2], on an abstract syntax tree representations of the system [10], or on isomorphic program dependence graphs [11]. Burd and Bailey give a comparison between the different approaches in [12]. Most of these approaches offer graphical user interfaces using dot plots to visualize the code clones. We worked with the Gemini environment for CCFinder [13] and with Duploc [14]. We found that the dot plot visualization technique was most useful for smaller fragments of source code but did not scale well for large systems such as Mozilla.

Casazza *et al.* describe the application of code clone detection tools on a large scale multi-platform software system in [15]. They explore the cloning percentages across different platform–dependent modules of the Linux kernel. The percentage of cloning that has been detected can be considered low. Compared to their approach we focus on the effect of code clones on change couplings.

Recent studies have shown why and how programmers introduce code clones into software systems [16] and how software development could benefit from the

inclusion of code clone detection tools into the development process [17]. The evolution of code clones has been investigated by Kim *et al.* in [18]. They provide a classification for evolving code clones but not for their impact on the change coupling behavior of the whole system. The main result of their paper is that code duplications cannot be considered inherently bad and do not need to be refactored in every case.

Work on the classification of code clones has recently been done by Kapser and Godfrey [19]. They propose a tool to interpret and classify the results gathered by code clone detection tools. Their case study also shows improvements in the elimination of false positive candidates returned by most clone detection tools.

The concept of the release history database (RHDB) was first described in [20] and [21]. The database uses version and bug tracking data and contains data obtained from the Mozilla open source project which uses CVS as version control system and Bugzilla for the organization of bug reports. Further work on logical and change couplings during the evolution of a software system was presented in [5, 22].

We adopted the visualization technique using polymetric views developed by Lanza and Ducasse [23] for the use with code clones and change couplings.

3 Framework

In this section we present our framework for analyzing the relation between code clones and change couplings. The framework consists of six steps as shown in Figure 1. The following subsections describe each step in detail.

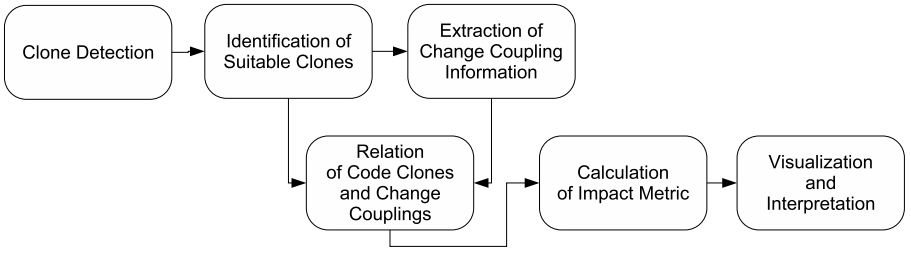


Fig. 1. Overview of the framework

3.1 Clone Detection

A pre-selection of three code clone detection tools has been made yielding the candidate tools Duploc [8], CloneDR [10], and CCFinder [9]. We selected the clone detection tool according to several criteria which we considered important for their applicability to the case study: language support for C and C++, maximal input size in lines of code, user interface, output format, recall, and precision. Most of these criteria are not directly measurable or even depend on the subjective perception of the user.

By means of these criteria we have chosen CCFinder as the clone detector most appropriate for our needs. Its recall, user interface, and output format fit best our needs to address our research goal. For an in-depth evaluation of the three clone detection tools we refer to [24].

3.2 Identification of Suitable Clone Candidates

As the goal of this framework is to define the relation between duplicated code and change couplings between files, the interesting clone pairs are those in which the cloned code fragments appear in two or more files. Furthermore, clones whose length varies or that appear or disappear during the examined period are considered more interesting than duplications that remain stable. This selection criterion is based on the assumption that there is a significant relation between code clones and change couplings.

Absolute numbers are inadequate when comparing different files because their lengths vary. The same applies to the length of cloned code fragments. Therefore our model of code clone classification relies on the clone coverage in every single file. This ratio is for file A compared to B defined as

$$CloneCoverage(A, B) = \frac{ClonedLines(A, B)}{NCLOC(A)}$$

where $ClonedLines(A, B)$ is the number of lines in file A that are clones of lines in file B. $NCLOC(A)$ is the number of lines of source code in file A not counting comments and blank lines. A cloned line is only counted once even if it is part of more than one clone pair or is covered multiple times by overlapping clones. When more than two files are compared, every pair of files out of this set has to be compared separately.

Two files A and B can share more than one semantically distinct clone pair. The types can be used to classify every instance of a clone pair. And in this paper, $CloneCoverage(X, Y)$ is always calculated for all code clones shared by X and Y.

To apply clone coverage to a set of evolving files, it is necessary to observe the clone coverage values over several versions of the files. These comparisons allow us the classification of each file pair sharing code clones into one of the following five types depending on the development of its clone coverage.

- **Type 0 (stable):** The relative length of cloned fragments in question remains the same between versions i and $i + 1$:

$$CloneCoverage(A, B)_i = CloneCoverage(A, B)_{i+1} \neq 0$$

- **Type 1 (new):** A clone is newly introduced in version $i + 1$:

$$CloneCoverage(A, B)_i = 0 \quad \wedge \quad CloneCoverage(A, B)_{i+1} > 0$$

- **Type 2 (removed):** A clone is removed between the versions i and $i + 1$:

$$CloneCoverage(A, B)_i > 0 \quad \wedge \quad CloneCoverage(A, B)_{i+1} = 0$$

– **Type 3 (increased):** Clone with increasing significance, *i.e.*, the clone coverage in version $i+1$ is larger than in version i:

$$CloneCoverage(A,B)_i < CloneCoverage(A,B)_{i+1}$$
$$\text{and} \quad CloneCoverage(A,B)_i > 0 \ \wedge \ CloneCoverage(A,B)_{i+1} > 0$$

– **Type 4 (decreased):** Clone with decreasing significance, *i.e.*, the clone coverage in version $i+1$ is smaller than in version i:

$$CloneCoverage(A,B)_i > CloneCoverage(A,B)_{i+1}$$
$$\text{and} \quad CloneCoverage(A,B)_i > 0 \ \wedge \ CloneCoverage(A,B)_{i+1} > 0$$

Types 1 to *4* indicate changes in code clones during evolution. Among them, those best suited for further investigation are clones of *Type 1* and *2*. We expect that the change couplings between files containing cloned fragments of each other show a relation between the changing code clones and their later couplings. If this assumption holds, for example two files into which a *Type 1* clone is introduced after version i are expected to exhibit more change couplings in subsequent versions. *Type 0* clones are also of interest because according to the hypothesis, change couplings caused by code clones are expected to be stable.

3.3 Extraction of Change Coupling Information

For this step of the framework we relied on our previous work on the release history database (RHDB) described in [20]. The RHDB contains data obtained, for example, from the Mozilla open source project. In particular, it stores data about the modification reports obtained from versioning control systems (CVS) repository of Mozilla and problem report data obtained from Mozilla's Bugzilla database.[1] In our framework we can exploit the RHDB to retrieve the change coupling data for the files that share code clones.

The number of change couplings between a pair of files (or similar entities of source code) during a given interval is the same for each file of a change coupled pair. The number of check–ins during the same time interval can, however, vary giving us a distinct ratio for each file. The coupling coverage metric we subsequently use is defined as

$$CouplingCoverage(A,B,I) = \frac{ChangeCouplings(A,B,I)}{Checkins(A,I)}$$

where $ChangeCouplings(A,B,I)$ is the number of times files A and file B are checked in together during time interval I and $Checkins(A,I)$ is the total number of times file A is checked in during I.

[1] http://bugzilla.mozilla.org

3.4 Relation of Code Clones with Change Couplings

A potential relation between code clones and change couplings is based on the assumption that pairs of source files sharing code clones are changed together [6]. This assumption has been taken for granted but not yet been proven.

For the investigation of this assumption we use the code clone and change coupling coverage values of each file pair and relate them. Results are represented in a dot plot where each dot refers to a file pair sharing code clones. The position of a dot is computed by drawing the code clone coverage value on the X-axis and the change coupling coverage value on the Y-axis.

Based on the assumption stated before we expect a concentration of dots along the diagonal meaning that low clone coverage leads to few change couplings and high clone coverage leads to frequent change couplings. An example of such a dot plot is depicted by Figure 3 in Section 4. And, as will be shown in the case study, the expectation is not always fulfilled.

To enable an interpretation of resulting dot plots we use regression analysis to quantify the relation between code clones and change couplings. In this paper we consider linear and logarithmic regression analysis. Two premises must be fulfilled for the regression to be significant. One is that a representative sample of files containing code clones is available for the calculation. The second is that this sample can be described with sufficient precision by a regression function, meaning that the correlation coefficient is close to 1.

3.5 Definition of a Metric to Describe the Impact of Code Clones

The relation presented before is based on the relative values of code clone and change coupling coverage. In addition the absolute length of a clone as well as the total number of change couplings influence our impact metric. That means, a longer fragment of duplicated code tends to have a larger influence than a shorter sequence. Furthermore, a file that is changed more often has a bigger potential of presenting a problem than a file that is never touched during the evolution of a system.

Based on these assumptions we select the following input parameters for the calculation of our impact metric:

- Clone coverage,
- Coupling coverage,
- Length of cloned fragments, and
- Absolute number of coupled check–ins.

Because of the difficulty to express the four parameters in one view a light–weight approach is used applying Lanza's polymetric views [23]. The key idea of polymetric views is to map metric values to graphical attributes, such as shape, size, and color of glyphs to activate the visual recognition capabilities of humans.

In our visualization, the four metrics listed above can be displayed in a Cartesian coordinate system enriched with additional use of color and the diameter of circles in the chart. The mapping of metric values to graphical attributes is depicted Figure 2.

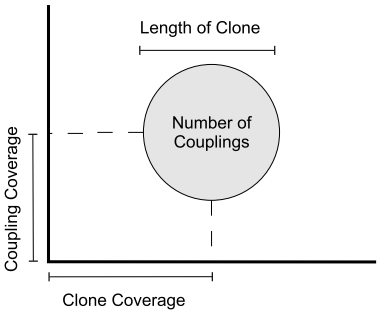


Fig. 2. Description of the metrics used in the visualization

The size of a circle is defined in proportion to the length of the clones. The maximum diameter is fixed and corresponds to the length of the longest clone. All other diameters are calculated proportionally to the length of the rest of the clones:

$$Diameter(A) = MaxDiameter \cdot \frac{ClonedLines(A,B)}{max(ClonedLines(X,Y))}$$

where $MaxDiameter$ is a constant describing the maximal diameter of a circle and $max(ClonedLines(X,Y))$ is the maximum length of cloned fragments to be visualized.

The fill color of a circle is defined in a way that the highest number of couplings is displayed as red. The intermediate colors are determined by variations of the RGB value proportional to the relative number of couplings so that a gradual transition to blue is achieved, which corresponds to zero couplings. The R and B–values are calculated by

$$R = \frac{ChangeCouplings(C,D,I)}{max(ChangeCouplings(X,Y,I))} \cdot 255, \quad \text{and } B = 255 - R$$

where R is the RGB–value for red and B the RGB–value for blue of the color of the circle in the chart. C and D are the specific files under consideration. $max(ChangeCouplings(X,Y,I))$ represents the maximal number of change couplings between any two files X and Y during interval I.

Unlike a numerical approach, this visualization is not dependent on a significant regression. The user is able to see possible problems and to react by closer inspection of the affected files.

4 Framework Validation

For the validation of our framework we applied the tools and methods to the open source project Mozilla[2]. The following sections report on our experiences and present the results of the experiments and the insights gained.

[2] http://www.mozilla.org/

4.1 Clone Detection

For the detection of code clones we selected the CCFinder tool. In our framework these output files form the basic input to calculate the correlations between code clones and change couplings.

The input data for our clone detection comprised the seven source code releases of Mozilla: 0.92, 0.97, 1.0, 1.3a, 1.4, 1.6, and 1.7. The release period between these releases is about 6 months. For each release we selected the ".c" and ".cpp" files that contain most parts of the implementation. We also skipped the header (".h") files because these files mostly contain only declarations and no implementation of functionality. The following description of the case study is based on the input data of Release 1.7 comprising 5,882 files with 2,980,000 lines of code (LOC).

For the configuration of the CCFinder tool we performed a number of test runs and came up with two possible configurations for the minimal length of code clones which are 30 and 70 tokens. 70 tokens were used when processing large amounts of data, such as all files of one release, to allow us to visualize the code clones. Otherwise, for our analysis of the relation between code clones and change coupling we used 30 tokens. Using 30 tokens results in code clones with a minimal length of 2.9–3.9 lines of C or C++ source code. Figure 3 shows the dot plot of detected code clones in source files of Mozilla Release 1.7 generated with the Gemini tool [13].

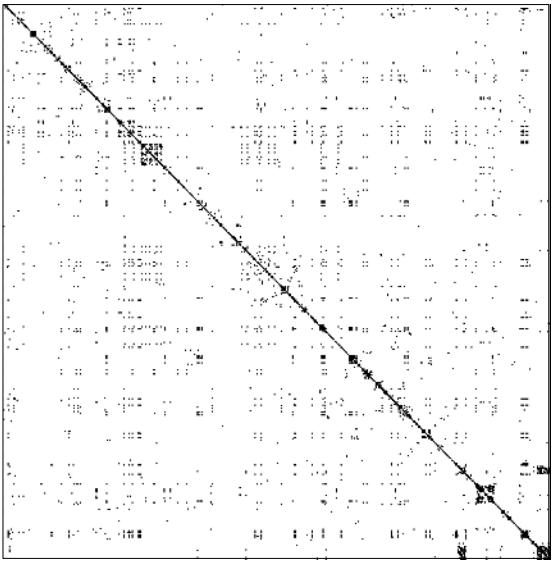


Fig. 3. Dot plot of code clones in Mozilla Release 1.7 (70 tokens)

In total the CCFinder tool detected 661,861 code clones in the source files of Release 1.7. In the dot plot files are arranged on a directory-basis allowing us to identify inter- and intra-module clones. Code clones within modules are indicated by the clusters positioned around the diagonal line. For instance, the cluster in the lower right corner shows the code clones within the "GFX and Widget–Mac" module. The other clusters in the dot plot indicate inter-module code clones.

4.2 Identification of Suitable Clone Candidates

Not all code clone candidates that are detected by CCFinder can be used for the purpose of this case study. One problem are false positives or clones only consisting of sequences of `#include`–statements, declarations of variables, or `switch`–statements. For the relation between code clones and change couplings *Types 1* to *4* are of interest because they changed during the evolution.

We selected a representative sample of 31 files to examine the types of clone containing file pairs occurring in the case study. These files form 21 file pairs of which almost none are of a single type of file pair within the examined interval. *Type 0* file pairs occur in 13, *Type 1* and *2* in 2, *Type 3* in 11, and *Type 4* in 12 pairs. Noteworthy, example pairs for *Type 0* and *4* file pairs are {`nsMathMLmoverFrame.cpp`, `nsMathMLmunderoverFrame.cpp`}, and {`os2/nsFilePicker.cpp`, `windows/nsFilePicker.cpp`} respectively.

Since in this case study most of the clone pairs occur in various types of file pairs, we did not consider the selection of special clone candidates. Therefore, we input all detected clones to the relation analysis.

4.3 Extraction of Change Coupling Information

For the extraction of the change coupling information we considered all files of Mozilla Release 1.7 that share at least one code clone. With this set of files we accessed our release history database (RHDB) and retrieved the change coupling information.

There is one major difference between the examination of code clones and that of change couplings that we have to consider: code clones involve the exploration of files at a given point in time – the date of each release of Mozilla – while the latter must be investigated over a certain time interval (*i.e.*, between two or more Mozilla releases).

Summarized we retrieved 139,523 change coupling records from the RHDB for all seven Mozilla releases (up to Mozilla 1.7).

4.4 Relation of Clone Data and Change Couplings

For relating the detected code clones with extracted change couplings we computed per file pair the code clone coverage and the change coupling coverage (see Section 3). The two coverage values then were plotted against each other yielding to the dot plot shown in Figure 4.

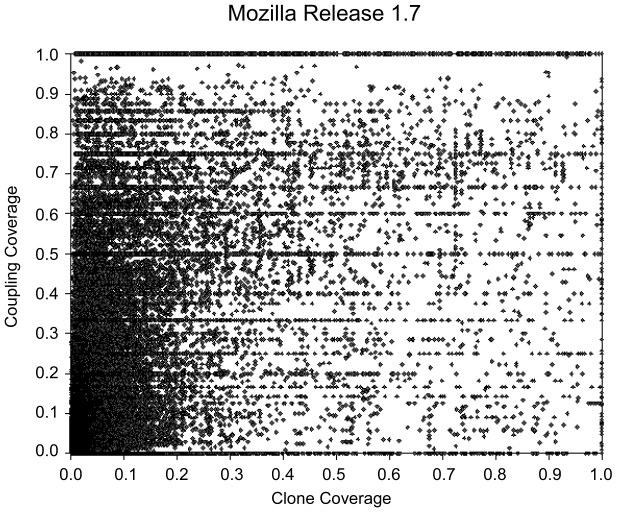


Fig. 4. Relation between code clones and change couplings in Mozilla 1.7

The concentration of values where the clone coverage ratio is below 0.2 indicates that the relation between clone and coupling coverage is pretty much random and difficult to interpret.

To prove the relation between code clones and change couplings we applied two types of (straightforward) regression analysis: linear and logarithmic regression analysis; using other functions are subject of future work. Because of the huge amount of data we started the regression analysis with three random samples of 65,536 file pairs sharing code clones. In each case, the R^2 value was better for linear than for logarithmic regression over the same sample. A linear regression resulted in the best fitting function with the highest coefficient of determination of 0.702:

$$CouplingCoverage(A, B, I) = 1.038 \cdot CloneCoverage(A, B) + 0.097$$

The resulting equation explains 70.2 % of the scattering visible in the chart. The other attempts at regression analysis yielded lower R^2 values.

Similar regression analyses were computed for the 30,433 instances of data where the clone coverage exceeded the threshold of 0.2. A linear regression with a low R^2 value of 0.2088 resulted in the equation

$$CouplingCoverage(A, B, I) = 0.512 \cdot CloneCoverage(A, B) + 0.4781$$

which is not a close fit compared to the results obtained by a sample of all input values.

The findings of our analysis indicate a certain relation between cloned fragments of source code and change couplings during evolution of the software. This connection was expected from previous work starting with [1]. Usually the larger

the clone coverage between two files is, the more often these files are coupled. However, based on our regression analyses it is neither possible to conclude that code duplications are reflected in high change coupling coverage values nor is the opposite true. From the results of this case study it is impossible to definitely exclude the possibility that there is in fact no statistically relevant correlation between code duplications and change couplings.

Change couplings can have causes other than code clones. Files fulfilling similar roles in the system often are changed together even though they might not contain many duplicated code fragments. Despite these exceptions, the general tendency for files with a high clone coverage value is to be coupled more often than files with a lower percentage of duplications.

An examination of clone and coupling coverage can be used to identify groups of files that would benefit from a determined refactoring effort. In Mozilla, we identified several such candidates. An example is the file `nsMathMLmsubFrame.cpp` of the MathML module which is coupled with files `nsMathMLmsubsupFrame.cpp` and `nsMathMLmsupFrame.cpp` in the same folder every single time it is changed between Releases 0.92 and 1.7.

Using our relation analysis it is not possible to distinguish harmless from dangerous code duplications simply by looking at the results of a code clone detection run on only one release of a software system. It is, however, safe to say that the larger the clone coverage is, the higher is the probability of it becoming "dangerous" during evolution. To express this degree of "danger" we applied our visualization technique presented in Section 3.

4.5 Visualizing the Impact of Code Clones

Because of the difficulties of establishing a sound mathematical correlation between code duplications and change couplings we applied the polymetric views visualization technique described in Section 3. This provided us with more insights into the relation and in particular pointed out file pairs with a strong relation being the candidates for a refactoring.

Figure 5 and Figure 6 depict the polymetric views created for the two modules MathML and JPEG of Mozilla Releases 0.9.2 and 1.7.

The MathML module consists of 26 C++ files between which 470 distinct pairs of files share duplicated code. Figure 5 depicts the situation for MathML. Both, the left and the right chart point out the 5 files E, F, G, H, and I in the upper right corner. They are frequently coupled with other files and share large fragments of duplicated code as indicated by the size of the circles. Files containing relatively few clones and with low code clone and change coupling coverage are drawn on the left side of the chart. By comparing the charts of both releases we distinguish the different types of (clone containing) file pairs (see Section 3.2).

For instance, the total length of clones as well as the clone coverage in L significantly decreased from Release 0.9.2 to 1.7 indicating a reengineering effort. According to our classification this is a good example for a *Type 4* file pair. Further *Type 4* file pairs are B, E, J, and K. In contrast, F represents a good

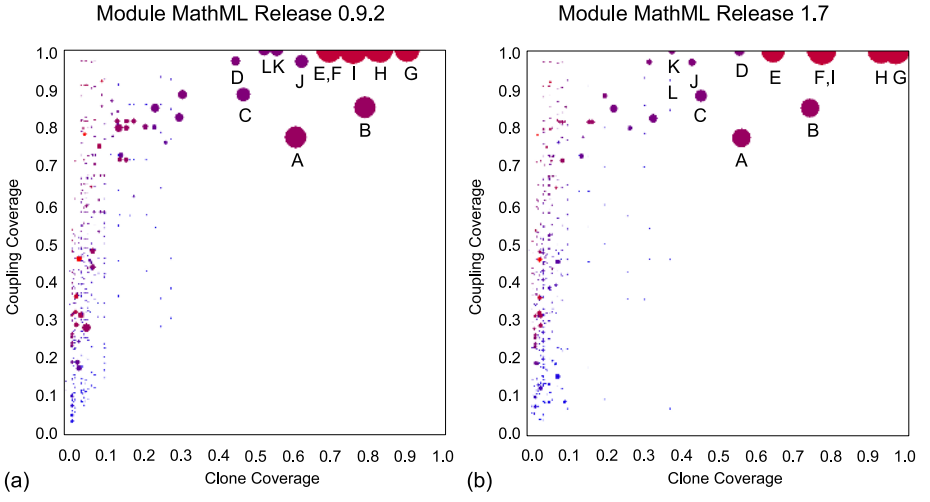


Fig. 5. Visualization of Mozilla module MathML of Releases 0.9.2 (a) and 1.7 (b)

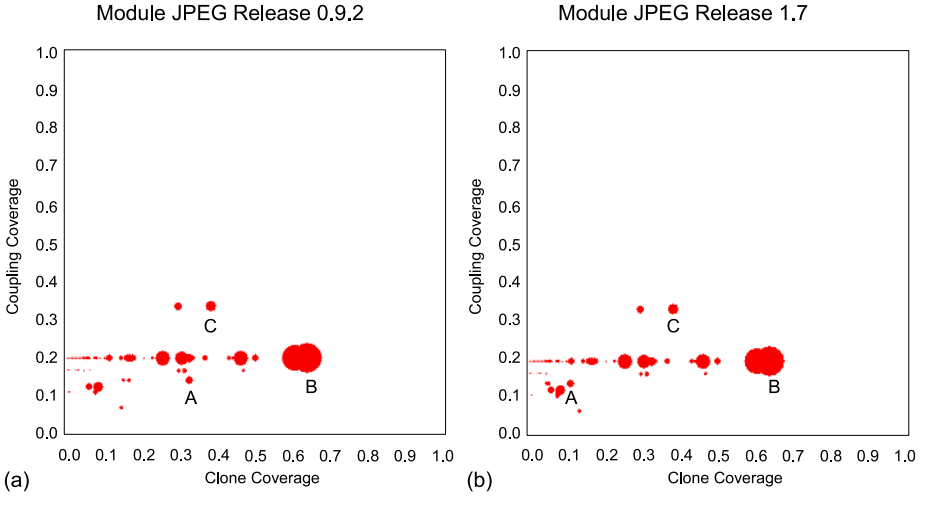


Fig. 6. Visualization of Mozilla module JPEG of Releases 0.9.2 (a) and 1.7 (b)

example of a *Type 3* file pair as indicated by an increasing clone coverage value. A similar trend can be seen for the file pairs represented by *A*, *C*, *D*, *G*, *H*, and *I*.

The situation for Mozilla's JPEG module is different as depicted by Figure 6. The 52 C files of this module form 230 distinct file pairs sharing code clones. In both charts most of the circles are equally red (dark) because every file of the module was coupled exactly once during the observed time periods. In this case,

the selection of candidates for a refactoring relies on the length of code clones and the clone coverage alone. The glyphs in both graphs show a large number of *Type 0* file pairs, for example B and C. Furthermore, there are few other types of file pairs, such as A, showing a relatively stable module JPEG.

Summarized, based on the metric values of file pairs sharing clones our polymetric views allowed us to spot the degree of "danger" of code clones. The most "dangerous" code clones were highlighted pointing us to the candidates in which code clones resulted in high change couplings, *e.g.*, H and G in Figure 5. They are first-class candidates to refactor.

5 Conclusions and Future Work

It is broadly assumed that the more clones two files share, the more often they have to be changed together. We address this problem of qualifying change couplings via code clone analysis.

In this paper, we discussed the relation of code clones and change couplings taken from release history data to examine whether a correlation exists between the two. For that, we proposed a framework to examine code clones and relate them to change couplings. The individual steps include clone detection and classification of clones into clone types, extraction of change couplings for the files in which the clones exist, calculating the relation between clones and change couplings, and computing and visualizing a relation metric to identify restructuring candidates.

We validated our framework with the open source project Mozilla and the results of the validation show that although the relation is statistically indeterminable it derives a reasonable number of cases where such a relation exists.

Our framework is not limited to the Mozilla case study; it is essentially independent of the type of system or of the programming language in which the system is written. The metrics defined are relatively simple yet effective to compute and require access to the system's source code and to a release history database containing release, modification, and bug report data.

We use polymetric views as a visualization technique to detect problematic code clones. This allows one to spot where a correlation between cloning and change coupling exists and, as a result, which files should be restructured to ease further evolution. If such a framework is integrated into a software engineering environment, it could potentially offer a useful guidance in the decision which clones are to be refactored. This is subject of our current work.

A result of this paper is that at least in the Mozilla case study, the correlation between code clones and change couplings is too complex to be expressed easily. For a significant distinction between clones that are irrelevant to the evolution of a system and clones that are harmful, more information is needed than what can be obtained automatically. Despite sophisticated tools that are available, the judgement of the software engineer is still needed.

As future work we plan to further improve the examination and visualization of the relation between code duplications and change couplings to distill all those

parts of a system in which clones are the cause for change couplings. We will further integrate this kind of analysis with our other evolution analysis tools to enable a more comprehensive picture by combining change dependencies, bugs, and code clones.

References

1. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
2. Mayrand, J., Leblanc, C., Merlo, E.: Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In: Proceedings of the International Conference on Software Maintenance (ICSM), Monterey, CA, USA, IEEE CS (1996) 244–253
3. Grubb, P., Takang, A.A.: Software Maintenance – Concepts and Practice. 2nd edn. World Scientific (2003)
4. Lehman, M.M., Belady, L.: Program Evolution Processes of Software Change. Academic Press (1985)
5. Gall, H., Hajek, K., Jazayeri, M.: Detection of Logical Coupling based on Product Release History. In: Proceedings of the 14th International Conference on Software Maintenance (ICSM), Bethesda, Maryland, USA, IEEE CS (1998) 190–198
6. Demeyer, S., Ducasse, S., Nierstrasz, O.: Object-Oriented Reengineering Patterns. Morgan Kaufmann, San Francisco, CA, USA (2003)
7. Baker, B.S.: A Program for Identifying Duplicated Code. Computing Science and Statistics **24** (1992) 49–57
8. Ducasse, S., Rieger, M., Demeyer, S.: A Language Independent Approach for Detecting Duplicated Code. In: Proceedings of the 15th International Conference on Software Maintenance (ICSM), Oxford, England, UK, IEEE CS (1999) 109–118
9. Kamiya, T., Kusumoto, S., Inoue, K.: CCfinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. IEEE Transaction on Software Engineering **28** (2002) 654–670
10. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone Detection Using Abstract Syntax Trees. In: Proceedings of the 14th International Conference on Software Maintenance (ICSM), Bethesda, Maryland, USA, IEEE CS (1998) 368–377
11. Krinke, J.: Identifying Similar Code with Program Dependence Graphs. In: Proceedings of the 8th Working Conference on Reverse Engineering (WCRE), Stuttgart, Germany, IEEE CS (2001) 301–310
12. Burd, E., Bailey, J.: Evaluating Clone Detection Tools for Use during Preventative Maintenance. In: Proceedings of the 2nd International Workshop on Source Code Analysis and Manipulation (SCAM), Montreal, Canada, IEEE CS (2002) 36–43
13. Ueda, Y., Kamiya, T., Kusumoto, S., Inoue, K.: Gemini: Maintenance Support Environment Based on Code Clone Analysis. In: Proceedings of the 8th International Symposium on Software Metrics (METRICS), Ottawa, Canada, IEEE CS (2002) 67–76
14. Rieger, M., Ducasse, S.: Visual Detection of Duplicated Code. In: Workshop on Object-Oriented Technology, Brussels, Belgium, Springer-Verlag (1998) 75–76
15. Casazza, G., Antoniol, G., Villano, U., Merlo, E., Penta, M.D.: Identifying Clones in the Linux Kernel. In: Proceedings of the 1sr International Workshop on Source Code Analysis and Manipulation (SCAM), Florence, Italy, IEEE CS (2001) 90–97

16. Kim, M., Bergman, L., Lau, T., Notkin, D.: An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In: Proceedings of the International Symposium on Empirical Software Engineering (ISESE), Redondo Beach, CA, USA, IEEE CS (2004) 83–92
17. Lague, B., Proulx, D., Mayrand, J., Merlo, E.M., Hudepohl, J.: Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In: Proceedings of the 13th International Conference on Software Maintenance (ICSM), Bari, Italy, IEEE CS (1997) 314–323
18. Kim, M., Sazawal, V., Notkin, D.: An Empirical Study of Code Clone Genealogies. In: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE), Lisbon, Protugal, ACM Press (2005) 187–196
19. Kapser, C., Godfrey, M.W.: Aiding Comprehension of Cloning Through Categorization. In: Proceedings of the 7th International Workshop on Principles of Software Evolution (IWPSE), Kyoto, Japan, IEEE CS (2004) 85–94
20. Fischer, M., Pinzger, M., Gall, H.: Populating a Release History Database from Version Control and Bug Tracking Systems. In: Proceedings of the 19th International Conference on Software Maintenance (ICSM), Amsterdam, The Netherlands, IEEE, IEEE CS (2003) 23–32
21. Gall, H., Jazayeri, M., Krajewski, J.: CVS Release History Data for Detecting Logical Couplings. In: Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE), Helsinki, Finland, IEEE CS (2003) 13
22. Fluri, B., Gall, H.C., Pinzger, M.: Fine-Grained Analysis of Change Couplings. In: Proceedings of the 5th International Workshop on Source Code Analysis and Manipulation, Budapest, Hungary, IEEE CS (2005) 66–74
23. Lanza, M., Ducasse, S.: Polymetric Views – A Lightweight Visual Approach to Reverse Engineering. IEEE Transactions on Software Engineering **29** (2003) 782–795
24. Geiger, R.: Evolution Impact of Code Clones – Identification of Structural and Change Smells based on Code Clones. Master's thesis, University of Zurich (2005) http://seal.ifi.unizh.ch/da.

Author Index