LNCS 2984

Michel Wermelinge
Tiziana Margaria-S

Fundamen
to Softwar

7th International Conferen
Held as Part of the Joint Eu
on Theory and Practice of S
Barcelona, Spain, March/A

Lecture Notes in Computer Science 2984

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

Springer

Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

Michel Wermelinger
Tiziana Margaria-Steffen (Eds.)

Fundamental Approaches to Software Engineering

7th International Conference, FASE 2004
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2004
Barcelona, Spain, March 29 – April 2, 2004
Proceedings

Springer

Series Editors

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

Volume Editors

Michel Wermelinger
Universidade Nova de Lisboa
Departamento de Informática
2829-516 Caparica, Portugal
E-mail: mw@di.fct.unl.pt

Tiziana Margaria-Steffen
Universität Dortmund
Fachbereich Informatik, LS V, Geb. IV
44221 Dortmund
E-mail: tiziana@ls5.cs.uni-dortmund.de

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

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

ISSN 0302-9743
ISBN 3-540-21305-8 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 10990582 06/3142 5 4 3 2 1 0

Foreword

ETAPS 2004 was the seventh 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 (FOSSACS, FASE, ESOP, CC, TACAS), 23 satellite workshops, 1 tutorial, and 7 invited lectures (not including those that are specific to the satellite events).

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 that 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 independent 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 2004 was organized by the LSI Department of the Catalonia Technical University (UPC), 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)
ACM SIGACT, SIGSOFT and SIGPLAN

The organizing team comprised

Jordi Cortadella (Satellite Events), Nikos Mylonakis, Robert Nieuwenhuis, Fernando Orejas (Chair), Edelmira Pasarella, Sonia Perez, Elvira Pino, Albert Rubio

and had the assistance of TILESA OPC.

ETAPS 2004 received generous sponsorship from:

UPC, Spanish Ministry of Science and Technology (MCYT), Catalan Department for Universities, Research and Information Society (DURSI), IBM, Intel.

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

Ratislav Bodik (Berkeley), Maura Cerioli (Genoa), Evelyn Duesterwald (IBM, Yorktown Heights), Hartmut Ehrig (Berlin), José Fiadeiro (Leicester), Marie-Claude Gaudel (Paris), Andy Gordon (Microsoft Research, Cambridge), Roberto Gorrieri (Bologna), Nicolas Halbwachs (Grenoble), Görel Hedin (Lund), Kurt Jensen (Aarhus), Paul Klint (Amsterdam), Tiziana Margaria (Dortmund), Ugo Montanari (Pisa), Hanne Riis Nielson (Copenhagen), Fernando Orejas (Barcelona), Mauro Pezzè (Milan), Andreas Podelski (Saarbrücken), Mooly Sagiv (Tel Aviv), Don Sannella (Edinburgh), Vladimiro Sassone (Sussex), David Schmidt (Kansas), Bernhard Steffen (Dortmund), Perdita Stevens (Edinburgh), Andrzej Tarlecki (Warsaw), Igor Walukiewicz (Bordeaux), Michel Wermelinger (Lisbon)

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, and finally Springer-Verlag for agreeing to publish the ETAPS proceedings. This year, the number of submissions approached 600, making acceptance rates fall to 25%. Congratulations to all the authors who made it into the final program! I hope that all the other authors still found a way of participating in this exciting event and I hope you will continue submitting.

In 2005, ETAPS will be organized by Don Sannella in Edinburgh. You will be welcomed by another “local”: my successor as ETAPS Steering Committee Chair – Perdita Stevens. My wish is that she will enjoy coordinating the next three editions of ETAPS as much as I have. It is not an easy job, in spite of what Don assured me when I succeeded him! But it is definitely a very rewarding one. One cannot help but feel proud of seeing submission and participation records being broken one year after the other, and that the technical program reached the levels of quality that we have been witnessing. At the same time, interacting with the organizers has been a particularly rich experience. Having organized the very first edition of ETAPS in Lisbon in 1998, I knew what they were going through, and I can tell you that each of them put his/her heart, soul, and an incredible amount of effort into the organization. The result, as we all know, was brilliant on all counts! Therefore, my last words are to thank Susanne Graf (2002), Andrzej Tarlecki and Paweł Urzyczyn (2003), and Fernando Orejas (2004) for the privilege of having worked with them.

Leicester, January 2004

José Luiz Fiadeiro
ETAPS Steering Committee Chair

Preface

This volume contains the proceedings of the seventh FASE, International Conference on Fundamental Approaches to Software Engineering.

FASE 2004 took place in Barcelona, Spain, during March 29–31, 2004, as part of the 7th European Joint Conference on Theory and Practice of Software (ETAPS), whose aims, organization, and history are detailed in the separate foreword by José Luiz Fiadeiro.

It is the goal of FASE to bring together researchers and practitioners interested in the challenges in software engineering research and practice: new software structuring and scaling concepts are needed for heterogeneous software federations that consist of numerous autonomously developed, communicating and interoperating systems. In particular, FASE aims at creating an atmosphere that promotes a cross-fertilization of ideas between the different communities of software engineering and related disciplines. New quality assurance methods are needed to guarantee acceptable standards of increasingly complex software applications. Different component paradigms are under discussion now, a large number of specification and modeling language are proposed, and an increasing number of software development tools and environments are being made available to cope with the problems. At the same time research on new theories, concepts, and techniques is under way, which aims at the development of a precise and (mathematically) formal foundation.

The presented contributions involved both pragmatic concepts and their formal foundation, leading to new engineering practices and a higher level of reliability, robustness, and evolvability of heterogeneous software. FASE comprised:

- **invited lectures** by Serge Abiteboul on *Distributed Information Management with XML and Web Services* and by Gruia-Catalin Roman on *A Formal Treatment of Context-Awareness*;
- **regular sessions** featuring 22 papers selected out of a record number of 91 submissions that ranged from foundational contributions to presentation of fielded applications; and
- **tool demonstrations**, featuring 4 short contributions selected out of 7 high-quality tool submissions.

FASE 2004 was hosted by the Technical University of Catalonia (UPC), and, being part of ETAPS, it shared the sponsoring and support described by the ETAPS Chair in the Foreword. Like ETAPS, FASE will take place in Edinburgh next year.

Warm thanks are due to the program committee and to all the referees for their assistance in selecting the papers, to José Luiz Fiadeiro for mastering the coordination of the whole ETAPS, and to Fernando Orejas and the whole team in Barcelona for their brilliant organization.

Recognition is due to the technical support team: Martin Karusseit and Markus Bajohr at the University of Dortmund provided invaluable assistance to all the involved people concerning the Online Conference Service provided by METAFrame Technologies during the past three months. Finally, we are deeply indebted to Claudia Herbers for her first-class support in the preparation of this volume.

January 2004 Tiziana Margaria and Michel Wermelinger

Program Committee

Ralph-Johan Back (Åbo Akademi, Finland)
Jean Bézivin (Univ. of Nantes, France)
Maura Cerioli (Univ. di Genova, Italy)
Matthew Dwyer (Kansas State Univ., USA)
Reiko Heckel (Univ. of Paderborn, Germany)
Constance Heitmeyer (Naval Research Laboratory, USA)
Heinrich Hußmann (Ludwig-Maximilians-Univ. München, Germany)
Dan Craigen (ORA, Canada)
Serge Demeyer (Univ. of Antwerp, Belgium)
John Fitzgerald (Univ. of Newcastle upon Tyne, UK)
David Garlan (Carnegie Mellon Univ., USA)
Antónia Lopes (Univ. de Lisboa, Portugal)
Jeff Magee (Imperial College, UK)
Tiziana Margaria (Univ. of Dortmund, Germany (co-chair))
Tom Mens (Univ. de Mons-Hainaut, Belgium)
Mauro Pezzè (Univ. di Milano, Italy)
Gian Pietro Picco (Politecnico di Milano, Italy)
Ernesto Pimentel (Univ. de Málaga, Spain)
Gabriele Taentzer (Technische Univ. Berlin, Germany)
Michel Wermelinger (Univ. Nova de Lisboa, Portugal (co-chair))

Reviewers

Francesca Arcelli
Myla Archer
James M. Armstrong
Richard Atterer
Roswitha Bardohl
Luciano Baresi
Gilles Barthe
Ramesh Bharadwaj
Bart Du Bois
Victor Bos
Benjamin Braatz
Pietro Braione
Antonio Brogi
Jean Michel Bruel
Carlos Canal
Walter Cazzola
Alexey Cherchago
Michael Colon
Giovanni Denaro
Ralph Depke
Manuel Díaz
Gabriella Dodero
Fernando Dotti
Francisco Durán
Claudia Ermel
Hartmut Ehrig
Cristina Gacek
Stefania Gnesi
Paula Gouveia
Giovanna Guerrini
Ping Guo
Jan Hendrik Hausmann
Neil Henderson
Kathrin Hoffmann
Ralph Jeffords
Miguel Katrib
Paul Kelly
Markus Klein
Jochen Küster
Luigi Lavazza
Andreas Leicher
Elizabeth Leonard
Johan Lilius
Sten Loecher
Marc Lohmann
Antonio Maña
Leonardo Mariani
Vincenzo Martena
Jesús Martínez
Francisco Martins
Viviana Mascardi
Angelo Morzenti
Isabel Nunes
Julia Padberg
Alexander Petrenko
Guo Ping
Andreas Pleuss
Venkatesh P. Ranganath
Gianna Reggio
Marina Ribaudo
Robby
Simone Roettger
B.N. Rossiter
P.Y.A. Ryan
Filip Van Rysselberghe
Andrei Sabelfeld
Amer Saeed
Tim Schattkowsky
Hans Stenten
Sebastian Thöne
Davide Tosi
Antonio Vallecillo
Pieter Van Gorp
Vasco T. Vasconcelos
Hendrik Voigt
Sebastian Uchitel
Jef Wijsen
Kwok Cheung Yeung
Andy Zaidman
Steffen Zschaler
Albert Zündorf

Table of Contents

Distributed Information Management with XML and Web Services[*]

Serge Abiteboul[**]

INRIA-Futurs, LRI and Xyleme
Serge.Abiteboul@inria.fr

Abstract. XML and Web services are revolutioning the automatic management of distributed information, somewhat in the same way HTML, Web browser and search engines modified human access to world wide information. To illustrate, we present Active XML that is based on embedding Web service calls inside XML documents. We mention two particular applications that take advantage of this new technology and novel research issues in this setting.
This paper is based primarily on research at INRIA-Futurs in the Gemo Group, around XML and Web services (in particular, the Xyleme, Active XML and Spin projects).

1 Introduction

The field of distributed data management has centered for many years around the relational model. More recently, the Web has simplified a world wide (or intranet) publication of data based on HTML (the backbone of the Web) and data access using Web browsers, search engines and query forms. However, because of the inconvenience of a document model (HTML is a model of document and not a data model) and limitations of the core HTTP protocol, the management of distributed information remains cumbersome. The situation is today dramatically improving with the introduction of XML and Web services. The Extensible Markup Language, XML [32], is a self-describing semi-structured data model that is becoming the standard format for data exchange over the Web. Web services [37] provide an infrastructure for distributed computing at large, independently of any platform, system or programming language. Together, they provide the appropriate framework for distributed management of information.

From a technical viewpoint, there is nothing really new in XML and Web services. XML is a tree data model much simpler than many of its ancestors such as SGML. Web services may be viewed as a simplified version of Corba. Together, they are bringing an important breakthrough to distributed data management simply because they propose Web solutions that can be easily deployed and

[*] The research was partly funded by EC Project DBGlobe (IST 2001-32645), the RNTL Project e.dot and the French ACI MDP2P.

[**] Gemo Team, PCRI Saclay, a joint lab between CNRS, Ecole Polytechnique, INRIA and Université Paris-Sud.

used independently of the nature of the machine, the operating system and the application languages. XML and Web services do not solve any open problem but they pave the way for a new generation of systems and generate a mine of new problems to solve.

We first describe some key aspects of XML and Web services (Section 2).

In Section 3, we argue for Active XML [7], AXML in short, that consists in embedding calls to Web services in XML documents. AXML documents provide extensional information as well as intensional one, i.e., means to obtain more data. Intensional information is specified by calls to Web services. By calling the service one can obtain up to date information. AXML also provides control of the service calls both from the client side (pull) or from the server side (push).

We illustrate issues in distributed data management and the use of AXML through two particular applications. In Section 4, we consider content warehouses, i.e., warehouses of non-numerical data, in a Peer-to-Peer environment. In Section 5, we consider the management of personal data.

This paper is not meant as a survey of distributed data management but more modestly, as a survey of some works of the author around the use of XML and Web services for distributed data management. The author wants to thank all the people with whom he worked on these projects and in particular B. Amann, O. Benjelloun, S. Cluet, G. Cobéna, I. Manolescu, T. Milo, A. Milo, B. Nguyen, M.-C. Rousset and J. Widom.

2 XML and Web Services

XML is a new exchange format promoted by the W3C [36] and the industry. An XML document may be viewed as a labeled ordered tree. An example of an XML document is given in Figure 1. We will see in the next section that this document is also an Active XML document. XML provides a nice mediation model, i.e., a lingua franqua, or more precisely a syntax, that most pieces of software can or will soon understand. Observe that unlike HTML, XML does not provide any information about the document presentation. (This is typically provided externally using a style sheet.)

In an XML document, the nodes are labeled. The document may be typed with a declaration given in a language called XML schema [33]. If such a document typing is provided, the labeling provides typing information for pieces of information. For instance, the typing may request a *movie* element to consist of a *title*, zero or more *author*s and *reviews*. Some software that knows about the type of this document will easily extract information from it. In that sense, the labels provide both a type and semantics to pieces of information. The typing proposed by XML schema is very flexible. In some sense, XML marries the document world with the database world and can be used to represent documents, but also structured or semistructured data [2]. For instance, it allows to describe a relational database as well as an HTML page.

An essential difference between a data standard such as XML and a document standard such as HTML is the presence of structure that enables the use of

```
<directory>
  <movies>
    <director>Hitchcock</director>
    <movie> <title>Vertigo</title>
      <actor>J. Stewart</actor> <actor>K. Novak</actor>
      <reviews> <sc service=reviews@cine.com >Vertigo</sc></reviews>
    </movie>
    <movie> <title>Psycho</title>
      <actor>N. Bates</actor>
      <reviews> <sc service=reviews@cine.com >Psycho</sc></reviews>
    </movie> <sc service=movies@allocine.com >Hitchcock</sc>
  </movies>
</directory>
```

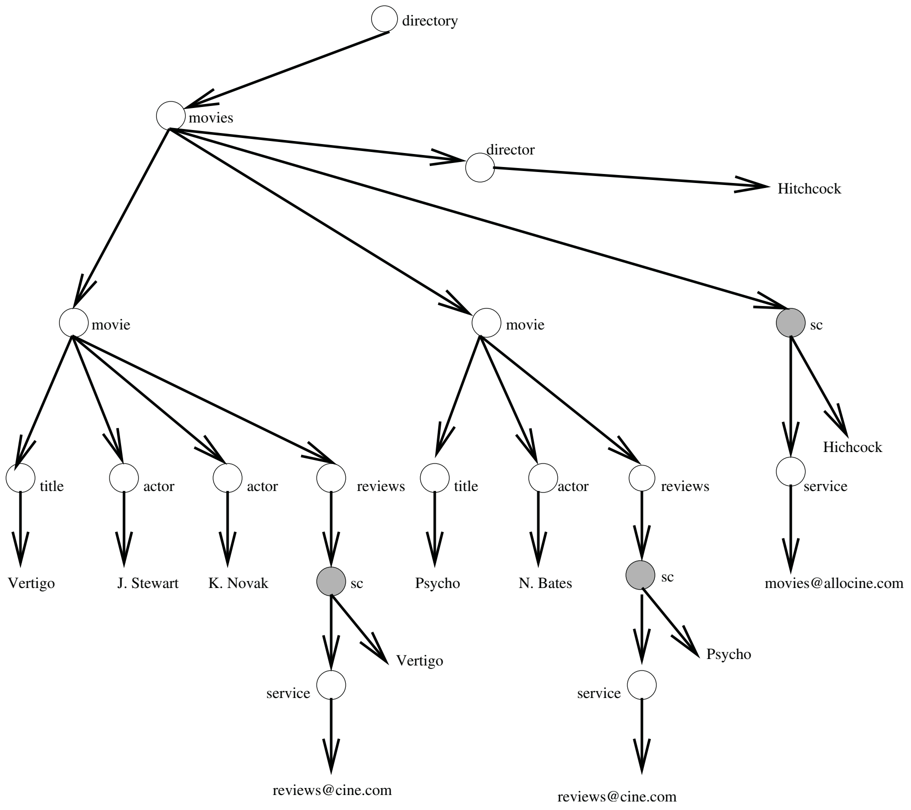


Fig. 1. Active XML document and its tree representation

queries beyond keyword search. One can query XML documents using query languages such as XPath or XQuery.

Web Services (successors of Corba and DCOM) are one of the main steps in the evolution of the Web. They allow active objects to be placed on Web sites providing distributed services to potential clients. Although most of the hype around Web services comes from e-commerce, one of their main current uses is for the management of distributed information. If XML provides the data model, Web services provide the adequate abstraction level to describe the various actors in data management such as databases, wrappers or mediators and the communications between them.

Web services in fact consist of an array of emerging standards. To find the desired service, one can query yellow-pages using UDDI [31] (Universal Discovery Description and Integration). Then to understand how to obtain the information, one use in particular WSDL [38] (Web Service Definition Language), something like Corba's IDL for the Web. One can then get the information with SOAP [30] (the Simple Object Access Protocol), an XML based lightweight protocol for exchange of information. Of course life is more complicated, so one also often has to sequence operations (see Web Services Choreography [39]) and consider issues such as access rights, privacy policy, encryption, payment, transactions, etc.

XML and Web services are nothing really new from a technical viewpoint. However, they form a nice environment to recycle old ideas in a trendy environment. Furthermore, they lead to an El-Dorado in terms of new problems and challenges for computer science research.

First, the trees are ordered and the typing more flexible. Tree automata [10], a technology rarely used in the context of data management, is a most appropriate tool for XML typing and query processing. Automata theory is perhaps enriching here with a new dimension the topic of descriptive complexity [16] that combines logic (to specify queries) and complexity (the resource required to evaluate queries).

Next, the distribution of data and computation opens new avenues at the frontier between data management (PODS-SIGMOD [26] like) and distributed computing (PODC [25] like). The distributed computation of a query may require opening connections between various query processors and cooperating towards answering the query, perhaps having partial computations move between different systems. Indeed, the context of the Web, a P2P environment with autonomous data sources, is changing dramatically the query evaluation problem:

– In classical distributed databases, we know about the structure of data in particular sources and the semantics of their content. On the Web, we typically discover new sources that we need to exploit. For instance, to integrate data sources [18], we need to understand their structure, possibly restructure them (tree rewriting) and build semantic bridges between their information (ontology mediation).
– In classical data management, we control data updates using technology such as transaction management or triggers. On the Web, data sources are autonomous and such control is unavailable. We may have to use services to synchronize copies of some data. We may also have to subscribe to change notifications. Often, the only solution is to poll regularly the data source to learn about changes. In most cases, we have to live with a lower consistency level.

Last but not least, we have to deal with the scale of the Web. The most spectacular applications of distributed data management are perhaps in that respect, Web search engines, e.g., Google, and Web look-up services, e.g., Kazaa. In both cases, the difficulty is the scaling to possibly billions of documents and millions of servers. This is leading to new algorithms. This also requires rethinking classical notions such as complexity and computability. For instance, how would one characterize the complexity of a query requiring a crawl of the entire Web? Linear? What would be the consistency of the result when a large number of the pages that have been visited no longer exist or have changed since our last visit.

3 Active XML

To illustrate the power of combining XML and Web services, we briefly describe Active XML that consists in embedding Web service calls in XML documents. This section is based on works in the context of the Active XML project [7].

In Active XML (AXML for short), parts of the data are given explicitly, while other parts consist of calls to Web services that generate more data. AXML is based on a P2P architecture. Each AXML peer acts as a client by activating Web service calls embedded in its documents. It also acts a server by supporting Web services corresponding to queries or updates over its repository of documents.

AXML is an XML dialect. For instance, the document in Figure 1 is an AXML document. The `sc` elements are used to denote embedded service calls. In this document, reviews are obtained from the `cine.com` Web site. Information about more Hitchcock movies may be obtained from the `allocine.com` site.

The data obtained by a call to a Web service may be viewed as intensional (it is originally not present). It may also be viewed as dynamic, in the sense of dynamic Web pages. The same call possibly returns different, up-to-date documents when called at different times. When a service call is activated, the data it returns is inserted in the document. Therefore, documents evolve in time as a consequence of service call activations. Of particular importance is thus the decision to activate a particular service call. In cases, this activation is decided by the peer hosting the document. For instance, a peer may decide to call a service only when the data it provides is requested by a user; the same peer may choose to refresh the data returned by another call on a periodic basis, say weekly. In other cases, the Web service provider may decide to send updates to the client, for instance because the latter registered to a subscription-based, continuous service.

A key aspect of the approach is that AXML peers exchange AXML documents, i.e., document with embedded service calls. Let us highlight an essential difference between the exchange of regular XML data and that of AXML data. In frameworks such as Sun's JSP or PHP, dynamic data is supported by programming constructs embedded inside documents. Upon request, all the code is evaluated and replaced by its result to obtain a regular, fully materialized HTML or XML document. But since Active XML documents embed calls to

Web services, one does not need to materialize all the service calls before sending some data. Instead, a more flexible data exchange paradigm is possible, where the sender sends an XML document with embedded service calls (namely, an AXML document) and gives the receiver the freedom to materialize the data if and when needed.

To conclude this section, we briefly mention two issues to illustrate the novel problems that are raised by the approach:

To call or not to call: Suppose someone asks for the information we have about the Vertigo movie. We may choose to call `cine.com` to obtain the reviews or not before sending the data. The decision may be guided by considerations such as performance, cost, access rights, security, etc. Now if we choose to activate the service call, it may return a document with embedded service calls and we have to decide whether to activate those or not, and so on, recursively. The solution we implemented is based on a negotiation between the peers to determine the type of data to exchange. This leads to complex automata manipulation [21]. Indeed, the general problem has deep connections with alternating tree automata, i.e., tree automata alternating universal and existential states [27].

Lazy service calls: As mentioned above, it is possible in AXML to specify that a call is activated only when the data it returns may be needed, e.g., to answer a query. A difficulty is then to decide whether a particular call is needed or not. For instance, if someone asks for information about the actors of "the 39 steps" of Hitchcock, we need to call `allocine.com` to get more movies of this director. Furthermore, if this service is sophisticated enough, we may be able to ask only for information about that particular movie ("push" the selection to the source). Some surprising connections between this problem and optimization techniques for deductive database and logic programming are exhibited in [6].

4 P2P Content Warehousing

In the next two sections, we mention two Web applications based on distributed information management, P2P content warehousing in this section and personal data management in the next. The content of this section is based on works in the Xyleme project [35] for content warehousing and on Spin [5] and MDP2P [20] for the P2P aspect.

Content Warehouse. Distributed enterprises and more generally communities centered around some common interest produce and need to share large volumes of data, e.g., reports, emails, contact addresses, documentation, contracts, Web sites. It is becoming more and more difficult to manage this information and critical to be able to find the desired data in a timely manner. This suggests organizing this information in *content warehouses* [4].

The goal of a warehouse [28] is to provide an integrated access to heterogeneous, autonomous, distributed sources of information. Queries are typically evaluated without access to the original sources. The focus here is on a warehouse of *content*, i,e., “qualitative information” and documents in various formats, rather than OLAP (on-line analytical processing) warehouses that are more concerned with “quantitative information”, that are typically organized in relations.

A content warehouse supports the construction, enrichment, monitoring and maintenance of large repositories of information with methods to access, analyze and annotate this information. Clearly, XML is the obvious candidate to represent the warehouse information, and in particular the meta-data about the warehouse documents, although some other formats such as pdf, html or doc, have also to be handled. Furthermore, since most systems will soon support Web service interfaces, Web services provide the natural infrastructure for communications in such a warehouse.

Peer to Peer Systems. There are many possible definitions of P2P systems. We mean here that a large and varying number of computers cooperate to solve particular tasks (here warehousing tasks) without any centralized authority. In other words, the challenge is to build an efficient, robust, scalable system based on (typically) inexpensive, unreliable, computers distributed on a wide area network. In the context of distributed data management, P2P is becoming more and more popular. See, for instance, [9,24,14,11,1,12].

The implementation of a content warehouse in a P2P setting may be motivated by a number of factors. For instance, from an economic viewpoint, a P2P system allows to share the costs and reduce them by taking advantage of existing infrastructures. Also, in a setting of a content warehouse with a very large volume of data and many users, we can take advantage of the distribution to improve performance.

The combination of the concept of warehouse (a centralized access to information) and of a P2P architecture (by definition stressing distribution) may seem a bit confusing, so let us try to articulate it more precisely. The data sources are heterogeneous, autonomous and distributed. The warehouse presents a unique entry point to this information, i.e., the warehouse is *logically* homogeneous and centralized. A P2P warehouse is an implementation of the concept of warehouse that is based on distributed peers. So, the warehouse is *physically* distributed over heterogeneous and autonomous machines. Note that a higher level of trust may possibly be achieved between the warehouse peers than between the original data sources.

From a technical viewpoint, the main issue is *distributed data management*, by no means a novel issue [22]. However, in a P2P environment, the context and in particular, the absence of central authority and the number of peers (from hundreds to possibly millions), change the rules of the game. Typically, problems that have long been studied take a different flavor. In particular, the following issues need to be addressed: (i) Information and service discovery (ii) Web crawling, (iii) document ranking (for queries) (iv) P2P mediation (integrating independent ontologies, e.g., [15,13,17]), (v) change monitoring.

In an on-going project called Spin (for set of pages of interest), we have worked on a centralized content warehouse. An XML repository is used to store the data and in particular the meta-data about the documents of the warehouse. Clearly, the management of meta-data is here an essential component, as advocated in Tim Berners-Lee's view of the Semantic Web [8]. All the processing is achieved via Web services, e.g., Web crawling, classification, tagging. A content warehouse in a particular domain may be specified using some graphical user interface. This specification is compiled into AXML documents where information to be acquired by the warehouse is described by Web service calls as well as the processing to be performed on this data. We are currently working on turning Spin into a P2P system.

5 Personal Data

The second Web application we consider here is distributed personal data management. This section is influenced by works in the DbGlobe project [23].

The management of personal data is becoming an important issue [19]. We typically handle more and more personal data in a variety of formats, distributed on a number of devices. In particular, we handle emails, addresses, bookmarks, documents, pictures, music, movies, bank accounts, etc. For instance, the information we have on a particular friend may be stored on a pda (address book, agenda), in phone directories (mobile or fix), in car repository (GPS coordinates, maps), in Web pages (her Web site), in pictures and movies, etc. Furthermore, lots of small personal data sources will soon be available in our environment, e.g., our house or our car.

All this personal data should be viewed as a distributed database. Indeed, to manage such data, we need most of the functionalities we mentioned for content warehouses; e.g., we need to query it or archive it. Furthermore, we need to maintain it up to date and in particular, we need to synchronize automatically various replicas of the same information.

Because of the standards they provide for distributed information management, XML and Web services form the appropriate infrastructure for doing this. Our thesis is that Active XML is the right articulation between the two to build more flexible systems for distributed personal data management.

One particular aspect of personal data management is that some of the devices are mobile, e.g., pda, car, phone. This has direct impact on aspects such as availability (a pda may be off line) or performance (a mobile phone has little storage and limited bandwidth). Also, some functionalities may depend on the location. For instance, to print a document, we need to find a nearby printer; and to select a restaurant, we care only for nearby places.

To mention another related application we are considering, consider the customization of a personal computer. It is typically is a cumbersome task that overwhelms most computer users. However, when considered from an abstract viewpoint, this is a simple task of data management. A particular person first needs access to its personal data. We already mentioned how this could be achieved. Then, the person is used to a particular environment and needs some specific software. If the specific needs of that person are described in an (A)XML document, we believe that the task of customizing the system to her could be handled automatically by the system. In particular, specific software the person uses could be obtained via Web services and maintained as well using Web service-based subscriptions.

Web services and XML only facilitate the exchange of information. The hard problems remain such as developing user friendly interfaces in this setting or automatically integrating and maintaining this information (including change control).

6 Conclusion

We have briefly discussed XML and Web services. We have illustrated how these could be used to facilitate the management of information distributed over a network such as the Web using Active XML and two applications, namely P2P content warehousing and personal data management. As already mentioned, there is a mine of new problems to study.

The management of structured and centralized data was made feasible by a sound foundation based on the development of relational database theory with deep connections to descriptive complexity. For the management of semistructured and distributed information, we are now at a stage where a field is building and a formal foundation is still in infancy. The development of this foundation is a main challenge for the researchers in the field.

References

1. K. Aberer, P-Grid: A Self-Organizing Access Structure for P2P Information Systems, CoopIS, 179-194, 2001.
2. S. Abiteboul, P. Buneman, and D. Suciu, Data on the Web, Morgan Kaufmann Publishers, 2000.
3. S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, T. Milo, Active XML Documents with Distribution and Replication, ACM SIGMOD, 2003.
4. S. Abiteboul, S. Cluet, G. Ferran and A. Milo, Xyleme Content Warehouse, Xyleme whitepaper, 2003.
5. S. Abiteboul, G. Cobena, B. Nguyen, A. Poggi, Construction and Maintenance of a Set of Pages of Interest (SPIN), Conference on Bases de Donnees Avancees, 2002.
6. S. Abiteboul and T. Milo, Web Services meet Datalog, 2003, submitted.
7. The AXML project, INRIA, http://www-rocq.inria.fr/verso/Gemo/Projects/axml.
8. T. Berners Lee, O. Lassila, and R. R. Swick, The semantic Web, Scientic American, vol. 5, 2001.
9. R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Seltzsam, K. Stocker, ObjectGlobe: Ubiquitous query processing on the Internet, The VLDB Journal, 10:48, 2001.

10. Tata, Tree Automata Techniques and Applications, H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison and M. Tommasi, www.grappa.univ-lille3.fr/tata/
11. F. M. Cuenca-Acuna, C. Peery, R. P. Martin, T. D. Nguyen, PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities, Department of Computer Science, Rutgers University, 2002.
12. T. Grabs, K. Böhm, H.-J. Schek, Scalable Distributed Query and Update Service Implementations for XML Document Elements, IEEE RIDE Int. Workshop on Document Management for Data Intensive Business and Scientific Applications, 2001.
13. F. Goasdoue and M.-C. Rousset, Querying Distributed Data through Distributed Ontologies: a Simple but Scalable Approach, 2003.
14. S. Gribble, A. Halevy, Z. Ives, M. Rodrig, D. Suciu, What can databases do for peer-to-peer? WebDB Workshop on Databases and the Web, 2001.
15. A. Y. Halevy, Z. G. Ives, D. Suciu, I. Tatarinov, Schema Mediation in Peer Data Management Systems, ICDE, 2003.
16. N. Immerman, Descriptive Complexity, Springer 1998.
17. A. Kementsietsidis, M. Arenas, and R. Miller, Mapping Data in Peer-to-Peer Systems: Semantics and Algorithmic Issues, ACM SIGMOD, 2003.
18. M. Lenzerini, Data Integration, A Theoretical Perspective, ACM PODS 20'02, Madison, Winsconsin, USA, 2002.
19. The Lowell Database Research Self-Assessment Meeting, 2003. research.microsoft.com/ gray/lowell/
20. Massive Data in Peer-to-Peer (MDP2P), a research project funded by the ACI Masses de Donnees of the French Ministry of Research, www.sciences.univ-nantes.fr/irin/ATLAS/MDP2P/
21. T. Milo, S. Abiteboul, B. Amann, O. Benjelloun, F. Dang Ngoc, Exchanging Intensional XML Data, ACM SIGMOD, 2003.
22. M.T. Özsszu, and P. Valduriez, Principles of Distributed Database Systems, Prentice-Hall, 1999.
23. E. Pitoura, S. Abiteboul, D. Pfoser, G. Samaras, M. Vazirgiannis, et al, DBGlobe: a service-oriented P2P system for global computing, SIGMOD Record 32(3): 77-82 (2003), DBGlobe is an IST research project funded by the European Community.
24. The Piazza Project, U. Washington, data.cs.washington.edu/p2p/piazza/
25. PODC, ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing.
26. PODS, ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems;
    SIGMOD, ACM SIGMOD Conference on the Management of Data.
27. Active Context-Free Games, L. Segoufin, A. Muscholl and T. Schwentick, 2003 (submitted)
28. J. Widom, Research Problems in Data Warehousing, In Proceedings of the 4th Int'l Conference on Information and Knowledge Management (CIKM), 1995.
29. F. M. Cuenca-Acuna, C. Peery, R. P. Martin, T. D. Nguyen, Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities, Department of Computer Science, Rutgers University, 2002.
30. Simple Object Access Protocol (SOAP) by Apache. ws.apache.org/soap/
31. The Universal Description, Discovery and Integration (UDDI) protocol, www.uddi.org/
32. The Extensible Markup Language (XML), www.w3.org/XML/

33. XML Typing Language (XML Schema), /www.w3.org/XML/Schema
34. XML Query Language (XQuery), www.w3.org/XML/Query
35. Xyleme Web site, www.xyleme.com
36. The World Wide Web Consortium (W3C), www.w3.org/
37. W3C Web Services Activity, www.w3.org/2002/ws/
38. Web Services Description Language (WSDL), www.w3.org/TR/wsdl
39. W3C Web Services Choreography Working Group,
www.w3.org/2002/ws/chor/

A Formal Treatment of Context-Awareness

Gruia-Catalin Roman, Christine Julien, and Jamie Payton

Department of Computer Science and Engineering
Washington University
Saint Louis, MO 63130
{roman,julien,payton}@wustl.edu

Abstract. Context-aware computing refers to a computing paradigm in which the behavior of individual components is determined by the circumstances in which they find themselves to an extent that greatly exceeds the typical system/environment interaction pattern common to most modern computing. The environment has an exceedingly powerful impact on a particular application component either because the latter needs to adapt in response to changing external conditions or because it relies on resources whose availability is subject to continuous change. In this paper we seek to develop a systematic understanding of the quintessential nature of context-aware computing by constructing a formal model and notation for expressing context-aware computations. We start with the basic premise that, in its most extreme form, context should be made manifest in a manner that is highly local in appearance and decoupled in fact. Furthermore, we assume a notion of context that is relative to the needs of each individual component, and we expect context-awareness to be maintained in a totally transparent manner with minimal programming effort. We construct the model from first principles, seek to root our decisions in these formative assumptions, and make every effort to preserve minimality of concepts and elegance of notation.

1 Introduction

Context-aware computing is a natural next step in a process that started with the merging of computing and communication during the last decade and continues with the absorption of computing and communication into the very fabric of our society and its infrastructure. The prevailing trend is to deploy systems that are increasingly sensitive to the context in which they operate. Flexible and adaptive designs allow computing and communication, often packaged as service activities, to blend into the application domain in a manner that makes computers gradually less visible and more agile. These are not new concerns for software engineers, but the attention being paid to context-awareness enhances a system's ability to become ever more responsive to the needs of the end-user or application domain. With the growing interest in adaptive systems and with the development of tool kits [1,2] and middleware [3] supporting context-awareness, one no longer needs to ponder whether context-aware computing is emerging as a new paradigm, i.e., a new design style with its own specialized models and support infrastructure. However, it would be instructive to develop a better understanding of how this transition took place, i.e., what distinguishes a design that allows a system to adapt to its environment from a design that could be classified as employing the context-aware paradigm. This is indeed the central question being addressed in this paper. We want to understand what context-aware computing is, and we do so by proposing a simple abstract conceptual model of context-awareness and by attempting to formalize it. Along the way, we examine the rationale behind our decisions, thus providing both a systematic justification for the model and the means for possible future refinements.

The term context-awareness immediately suggests a relation between some entity and the setting in which it functions. Let us call such an entity the *reference agent* – it may be a software or hardware component – and let us refer to the sum total of all other entities that could (in principle) affect its behavior as its *operational environment*. We differentiate the notion of operational environment from that of context by drawing a distinction between potentiality and relevance. While all aspects of the operational environment have the potential to influence the behavior of the reference agent, only a subset are actually relevant to the reference agent's behavior. In formulating a model of context-awareness we need to focus our attention on how this relevant subset is determined. To date, most of the research on context-awareness considers a restricted context, i.e., context is what can be sensed locally, e.g., location, temperature, connectivity, etc. However, distant entities can affect the agent's behavior, and the size of the zone of influence depends upon the needs of the specific application. The scope and quality of the information gathered about the operational environment affect the cost associated with maintaining and accessing context information. These suggest that a general model of context-awareness must allow an agent to work with a context that may extend beyond its immediate locality (i.e., support node or host) while also enabling it to control costs by precisely specifying what aspects of the operational environment are relevant to its individual needs as they change over time.

A model of context-awareness must be *expansive*, i.e., it must recognize the fact that distant entities in the operational environment can affect an agent's behavior [4]. This requirement states that one should not place a priori limits on the scope of the context being associated with a particular agent. While specific instantiations of the model may impose restrictions due to pragmatic considerations having to do with the cost of context maintenance or the nature of the physical devices, application needs are likely to evolve with time. As a consequence, fundamental assumptions about the model could be invalidated. To balance out the expansive nature of the model and to accommodate the need for agents to exercise control over the cost of context maintenance, we also require the model to support a notion of context that exhibits a high degree of *specificity*. In other words, it must be possible for context definitions to be tailored to the needs of each individual agent. Furthermore, as agents adapt, evolve, and alter their needs, context definitions also should be amenable to modification in direct response to such developments.

Expansiveness and specificity are central to achieving generality. They are necessary but not sufficient features of the context-aware computing paradigm that consider the way in which the operational environment relates to an agent's notion of context, i.e., the distinction between potentiality and relevance. They fail to consider the manner in which the agent forms and manipulates its own notion of context. The only way an agent can exercise control over its context is to have an *explicit* notion of context. This gives the agent the power to define its own context and to change the definition as it sees fit. It also formalizes the range of possible interactions between the agent and its operational environment. Consequently, context definition has to be an identifiable element of the proposed model and must capture the essential features of the agent/context interaction pattern. Separation of concerns suggests that an agent's context specification be *separable* from its behavior specification. The agent behavior may result in changes to the definition of context, but the latter should be readily understood without one needing to examine the details of the agent behavior. This requirement rules out the option of having to derive context from the actions of the agent. This distinction is important because many systems interact with and learn about their operational environment without actually employing the context-aware paradigm. Finally, context maintenance must be *transparent*. This implies that the definition of context must be sufficiently abstract to free the agent of the operational details of discovering its own context and sufficiently precise for some underlying support system to be able to determine what the context is at each point in time.

To illustrate our perspective on context-aware computing, let us examine the case of an application in which contextual information plays an important role, but the criteria for being classified as employing the context-aware paradigm are not met. Consider an agent that receives and sends messages, learns about what other agents are present in its environment through the messages it receives, and uses this information to send other messages. Indeed the context plays an important role in what the agent does, and the agent makes decisions based upon the knowledge it gains about its operational environment. More precisely, the agent implicitly builds an acquaintance list of all other agents in the region through a gossiping protocol that distributes this information, and it updates its knowledge by using message delivery failures that indicate agent termination or departure. However, we do not view this as an instance of the context-aware paradigm; the way the agent deals with the environment may be considered extensible but it is not specific, explicit, separable, or transparent. It is particularly instructive to note what is required to transform this message-passing application into one that qualifies as an instance of the context-aware paradigm.

Specificity could be achieved by having each agent exercise some judgment regarding which other agents should or should not be included in its acquaintance list. Explicitness could be made manifest by having the agent include a distinct representation of the acquaintance list in its code – a concrete representation of its current context. Separability could be put in place by having the code that updates the acquaintance list automatically extract agent information from arriving messages for placement in the acquaintance list, e.g., by employing the interceptor pattern. Transparency requires the design to go one step further by having the agent delegate the updating of the acquaintance list to an underlying support infrastructure; this, in turn, demands that the definition of the context be made explicit to the support infrastructure either at compile or at run time – an evaluation procedure to establish which other agents qualify as acquaintances and which do not suffices. The result is an application that exhibits the same behavior but a different design style; the agent's context is made manifest through an interface offering access to a data structure that appears to be local, is automatically updated, and is defined by the agent which provides the admission policy controlling which agents in the region are included or excluded from the list. This is not the only way to employ the context-aware paradigm but clearly demonstrates the need for the designer to adopt a different mind-set.

The principal objective of this paper is to explore the development of an abstract formal model for context-aware computing; no such model is available in the literature to date, other than a preliminary version of this work [5]. Because our ultimate goal is to achieve a better understanding of the essence of the context-aware computing paradigm, we seek to achieve minimality of concepts and elegance of notation while remaining faithful to the formative assumptions that define our perspective on context-awareness. The resulting model is called Context UNITY and has its roots in our earlier formal work on Mobile UNITY [6,7] and in our experience with developing context-aware middleware for mobility. Context UNITY assumes that the universe (called a system) is populated by a bounded set of agents whose behaviors can be described by a finite set of program types. At the abstract level, each agent is a state transition system, and context changes are perceived as spontaneous state transitions outside of the agent's control. However, the manner in which the operational environment can affect the agent state is an explicit part of the program definition. In this way, the agent code is local in appearance and totally decoupled from that of all the other agents in the system. The context definition is an explicit part of the program type description, is specific to the needs of each agent as it changes over time, and is separate from the behavior exhibited by the agent. The design of the Context UNITY notation is augmented with an assertional style proof logic that facilitates formal reasoning about context-aware programs.

The remainder of this paper is organized as follows. The next section presents our formalization of context-awareness in detail. In Section 3, we outline the proof logic associated with the model. Section 4 shows how the model can express key features of several existing context-aware systems. Conclusions appear in Section 5.

2 Formalizing Context-Awareness

Context UNITY represents an application as a community of interacting agents. Each agent's behavior is described by a program that serves as the agent's prototype. To distinguish agents from each other, each has a unique identifier. Because we aim to model context-aware systems, an agent must access its environment, which, in Context UNITY, is defined by the values of the variables other agents in the system are willing to expose. As described in the previous section agents require context definitions tailored to their individualized needs. In Context UNITY, agents interact with a portion of the operational environment defined through a unique set of variables designed to handle the agent's context needs.

A central aspect of Context UNITY is its representation of program state. Three categories of variables appear in programs; they are distinct in the manner in which they relate to context maintenance and access. First, a program's *internal variables* hold private data that the agent uses but does not share with the other agents in the system. They do not affect the operational environment of any other agent. *Exposed variables* store the agent's public data; the values of these exposed variables can contribute to the context of other agents. The third category of variables, *context variables*, represent the context in which the particular agent operates. These variables can both gather information from the exposed variables of other agents and push data out to the exposed variables of other agents. These actions are governed by context rules specified by each agent and subject to access control restrictions associated with the exposed variables.

In the remainder of this section, we first detail the structure of a Context UNITY system. We then show how programs use context variables to define a context tailored to the needs of each particular agent and the mechanics that allow an agent to explicitly affect its operational environment. Throughout we provide examples using the model to reinforce each concept.

2.1 Foundational Concepts

Context UNITY represents an application as a *system specification* that includes a set of *programs* representing the application's component types. Fig. 1 shows the Context UNITY representation of a **System**. The first portion of this definition lists programs that specify the behavior of the application's individual agents. Separating the programs in this manner encapsulates the behavior of different application components and their differing context needs. The **Components** section of the system declares the instances of programs, or agents, that are present in the application. These declarations are given by referring to program names, program arguments, and a function (*new_id*) that generates a unique id for each agent declared. Multiple instantiations of the same program type are possible; each resulting agent has a different identifier. The final portion of a system definition, the **Governance** section, captures interactions that are uniform across the system. Specifically, the rules present in this section describe statements that can impact exposed variables in all programs throughout the system. The details of an entire system specification will be made clearer through examples later in this section. First we describe in detail the contents of an individual Context UNITY program.

Each Context UNITY program lists the variables defining its individual state. The declaration of each variable makes its category evident (internal, exposed,

```
System SystemName
  Program ProgramName (parameters)
    declare
      internal — internal variable declarations
      exposed — exposed variable declarations
      context — context variable declarations
    initially — initial conditions of variables
    assign — assignments to declared variables
    context
      definitions affecting context variables—they can pull information from and
          push information to the environment
  end
  ... additional program definitions ...
  Components
    the agents that make up the system
  Governance
    global impact statements
end SystemName
```

Fig. 1. A Context UNITY Specification

or context). A program's **initially** section defines what values the variables are allowed to have at the start of the program.

The **assign** section defines how variables are updated. These assignment statements can include references to any of the three types of variables. Like UNITY and its descendants, Context UNITY's execution model selects statements for execution in a weakly-fair manner – in an infinite execution, each assignment statement is selected for execution infinitely often. In the assignment section, a program can use simple assignment statements, transactions, or reactions. A *transaction* is a sequence of simple assignment statements which must be scheduled in the specified order with no other (non-reactive) statements interleaved. They capture a form of sequential execution whose net effect is a large-grained atomic state change. In the **assign** section of a program, a transaction uses the notation: $\langle s_1; s_2; \ldots; s_n \rangle$. A *reaction* allows a program to respond to changes in the state of the system. A reaction is triggered by an enabling condition Q and has the form s **reacts-to** Q. As in Mobile UNITY, Context UNITY modifies the execution model of traditional UNITY to accommodate reactions. Normal statements, i.e., all statements other than reactions, continue to be selected for execution in a weakly-fair manner. After execution of a normal statement, the set of all reactions in the system, forming what we call a *reactive program*, executes until it reaches *fixed-point*. During the reactive program's execution, the reactive statements are selected for execution in a weakly-fair manner while all normal statements are ignored. When the reactive program reaches a fixed-point, the weakly-fair selection of normal statements continues.

In Context UNITY, an agent's behavior is defined exclusively through its interaction with variables. To handle context interactions, Context UNITY introduces context variables and a special **context** section that provides the rules that manage an agent's interaction with its desired context. Specifically, the

context section contains definitions that sense information from the operational environment and store it in the agent's context variables. The rules can also allow the agent to affect the behavior of other agents in the system by impacting their exposed variables. The use of this special **context** section explicitly separates the management of an agent's context from its internal behavior.

Two prototypical uses of the **context** section lie at the extremes of sensing and affecting context. First, a program's context definition may only read the exposed variables of other programs but not affect the variables' values. When used in such a way, we refer to the context variables as *sentient variables* because they only gather information from the environment to build the agent's context. In the other extreme case, a program can use its context variables to disperse information to components of the environment. From the perspective of the reference agent, this affects the context for other agents, and we refer to context variables used in this manner as *impact variables*. While these two extremes capture the behavior of context-aware systems in the most common cases, the generality of Context UNITY's context specification mechanism allows it to model a variety of systems that fall between these two extremes. The examples discussed in Section 4 demonstrate this in more detail.

The acquaintance list application introduced in the previous section provides a list of nearby coordination participants. Several context-aware systems in the literature, e.g., Limone [8], use this data structure as a basis for more sophisti-

```
System AcquaintanceManagement
  Program Agent1
    declare
      exposed id ! agent_id : agent_id
                  λ ! location : location
      context Q : set of agent_id
    assign
      ... definition of local behavior ...
    context
      define — define Q based on desired properties of acquaintance list members
  end
  Program Agent2
    declare
      exposed id ! agent_id : agent_id
                  λ ! location : location
      context Q : set of agent_id
    assign
      ... definition of local behavior ...
    context
      define — define Q based on different restrictions
  end
  Components
    Agent1[new_id], Agent1[new_id], Agent2[new_id]
end AcquaintanceManagement
```

Fig. 2. A Context-Aware System for Acquaintance Maintenance

cated coordination mechanisms. The acquaintance list is defined by dynamically changing needs of a reference agent. Fig. 2 shows a Context UNITY specification for an application that relies on the usage of an acquaintance list. This system consists of three agents of two differing types. Each agent stores its unique agent id in an exposed variable named agent_id that is available to other programs. Because we are modeling systems that entail agent mobility, each agent also has a variable named location that stores its location. The movement of the agent is outside this example; it could occur through local assignment statements to the location variable (in the **assign** section of the individual program) or even by a global controller (via the **Governance** section of the system). Both *id* and λ are local handles for built-in variables whose names are agent_id and location, respectively. We discuss these built-in variables in more detail later in this section. Each program type has individualized behavior defined via the **assign** section that may use additional context variables or definitions. In this example, we are most concerned with the maintenance of the acquaintance list. Each agent declares a context variable Q of type set that will store the contents of the acquaintance list. Different program types (in this case, *Agent1* and *Agent2*) employ different eligibility qualification criteria for the members of the acquaintance list, exemplified in the **context** section of each program type. This example shows a high-level definition of a context variable. In the acquaintance management specification, each program's **context** section contains a rule that describes how the context variable Q is updated. Later in this section we will show exactly what this rule entails. First however, we expound on the structure of Context UNITY exposed variables.

ι	the variable's unique id
π	the id of the owner agent
η	the name
τ	the type
ν	the value
α	the access control policy

Fig. 3. Variable Components

Exposed Variables Revisited. In UNITY and many of its descendants, variables are simply references to values. In Context UNITY, both internal and context variables adhere to this standard. However, references to exposed variable appearing in the program text are references to more complex structures needed to support context-sensitive access within an unknown operational environment. These handle names have no meaning outside the scope of the program. A complete semantic representation of exposed variables is depicted in Fig. 3. Each exposed variable has a unique id ι – uniqueness could be ensured by making each variable unique within an agent and combining this with the unique agent id. This unique id is used in the context interaction rules to provide a handle to the specific variable. The agent owning the exposed variable, π or type agent_id, also appears in the semantic structure and allows an exposed variable to be selected based on its owner. An exposed variable's name, η, provides information about the kind of data the variable contains; the name of an exposed variable can be changed by the program's assignment statements. The type τ reflects the exposed variable's data type and is fixed. An exposed variable's value, ν, refers to the data value held by the variable. Programs refer to the value when assigning to the variable or when accessing the value the variable stores. The value of an exposed variable can be assigned in the **assign** section or can be determined by a different program's impact on its environment. The program can control the extent to which its exposed variables can be modified by others using the access control policy described below.

Modeling Access Control. The final component of an exposed variable in Context UNITY, α, stores the variable's access control policy. Because many context-aware systems and applications use some form of access restriction, Context UNITY provides a generalized mechanism for modeling access control. An access policy determines access based on properties of the particular agent accessing the variable. The access control policy determines both the readability and writability of the particular variable on a per-agent basis. The function α takes as arguments credentials provided by the reference agent and returns the set of allowable operations on this variable, e.g., {r, w} signifies permission to both read and write the particular exposed variable. Because Context UNITY associates an access control policy with each variable, it models the finest-grained access restrictions possible in a context-aware application. This model can be tailored to the needs of current context-aware systems, including those that utilize a trusted third party for authentication.

Built-in Variables. To ease representation of context-aware interactions, Context UNITY programs contain four built-in exposed variables. In Context UNITY, these variables are automatically declared and have default initial values. An individual program can override the initial values in the program's **initially** section and can assign and use the variables throughout the **assign** and **context** sections. The first of these variables has the name "location" and facilitates modeling mobile context-aware applications by storing the location of the program owning the variable. This variable is exposed and available to other programs to use. An example use of this variable was shown in the system in Fig. 2. The definition of location can be based on either a physical or logical space and can take on many forms. This style of modeling location is identical to that used in Mobile UNITY. The second of Context UNITY's built-in variables is also exposed and has the name "type", and its value is the program's name (e.g., "*Agent1*" or "*Agent2*" in the example system). As we will see, the use of this variable can help context variables select programs based on their general function. The third of the built-in variables has the name "agent_id" and holds the unique identifier assigned to the agent when the agent is instantiated in the **Components** section. The final built-in variable is internal and has the local handle "*credentials*". It is used in Context UNITY interactions to support access control restrictions. Specifically, the variable stores a profile of attributes of the program that are provided to the access control policies of the exposed variables of other programs. These credentials are available to access control policies when determining whether or not this program has access to a particular exposed variable.

Context Specification. Context-aware applications rely on conditions in the environment for adaptation. Context UNITY facilitates specification of context interactions through the use of context variables that use the exposed variables of other agents to provide exactly the context that a reference agent requires. In a Context UNITY program, the **context** section of a program contains the rules that dictate restrictions over the operational environment to define the context over which an agent operates. Additionally, the rules in the **context** section allow the agent to feed back information into its context. Structuring the **context** section as a portion of each program allows agents to have explicit and individualized interactions with their contexts.

As indicated in the beginning of this section, due to the unpredictable nature of the dynamic environments in which context-aware agents operate, their context definitions require a mechanism to handle their lack of a priori knowledge about the operational environment. In Context UNITY, we introduce *non-deterministic assignment statements* to the definition of context. Specifically, the non-deterministic assignment statement $x := x'.Q$ assigns to x a value x' non-deterministically selected from all values satisfying the condition Q [9]. A program's context rules define how an agent can access and interact with the exposed variables of other agents. It can select which other agents' variables affect its behavior by employing non-deterministic assignments and existential quantification. The flexibility of this selection mechanism allows agents that contribute to the context to be selected based on attributes defined in their exposed variables. For example, in a mobile context-aware application, an agent can use the built-in Context UNITY location variable to store its current physical location. Whenever the component moves, the agent updates the location variable using an assignment statement in the local **assign** section. Another agent can use relative distance to identify which other agents are to contribute to its context. We refer to this selection of agents based on their properties as *context-sensitive program selection.*

Context UNITY wraps the use of non-deterministic assignment in a specialized notation for handing context-aware interactions. To manage its interaction with context information, a program uses statements of the following form in its **context** section:

c **uses** *quantified variables*
  given *restrictions on variables*
  where c **becomes** *expr*
    expr₁ **impacts** *exposed variable₁*
    expr₂ **impacts** *exposed variable₂*
    ...
  [**reactive**]

This expression, which we refer to as a *context rule*, governs the interactions associated with the context variable c. A context rule first declares existentially quantified dummy variables to be used in defining the interactions with the exposed variables that relate to the context variable c. The scope of these dummy variables is limited to the particular context rule that declares them. The expression can refer to any exposed variables in the system, but referring to other programs' exposed variables explicitly requires the program to have advance knowledge about the other components it will encounter over time, which programs rarely have. Typically, context-aware applications rely on opportunistic interactions that cannot be predetermined. To capture this style of interaction in Context UNITY, the exposed variables that contribute to the context rule are selected in a context-sensitive manner using the restrictions provided in the rule's definition. As one example, because a wireless context-aware application contains many agents that may or may not be connected, the restrictions used in a context rule for a particular application must account for the connectivity restrictions imposed by the operational environment.

Given the set of exposed variables selected in accordance with the restrictions, the context rule can define an expression, *expr*, over the exposed variables and any locally declared variables (internal, exposed, or context). The result of evaluating this expression is assigned to the context variable. The context rule can also define how this context variable impacts the operational environment.

The execution of each context rule can optionally be declared **reactive**, which dictates the degree of consistency with which the context rule reflects the environment. If a context rule is declared **reactive**, it becomes part of the system's reactive program that is executed to fixed-point after the execution of each normal statement. Using a reaction guarantees that the context information expressed by the rule remains consistently up to date because no normal statements can execute until the reactive program reaches fixed-point. If not declared **reactive**, the context rule is a normal, unguarded statement and part of Context UNITY's normal execution model.

Within a context rule, if no explicit restrictions are placed on the referenced exposed variables, two restrictions are automatically assumed. The first requires that the variable referenced be an exposed variable in its owner program since only exposed variables are accessible from other programs. The second implicit restriction requires that the program whose context uses a particular exposed variable must satisfy the variable's access control policy. Consider the following simple context rule that pulls the value out of some exposed variable, places the value in the context variable c, and deletes the value from the exposed variable used. The statement is a reactive statement that is triggered when a is larger than the value of some local variable x:

$$\begin{array}{ll} c\ \textbf{uses} & a \\ \quad\textbf{given} & a > x \\ \quad\textbf{where} & c\ \textbf{becomes}\ a \\ & 0\ \textbf{impacts}\ a \\ \quad\textbf{reactive} & \end{array}$$

This reactive construct makes the rule part of the system's set of reactive statements. This context rule corresponds to the following formal definition, which includes the two implicit restrictions on the exposed variable a as discussed above:

$$\begin{array}{l}\langle a: \ a = a'.(\mathsf{var}[a'] > x \wedge \{r, w\} \subseteq \mathsf{var}[a'].\alpha(\mathit{credentials})) \\ \quad :: (c := \mathsf{var}[a].\nu \ || \ \mathsf{var}[a].\nu := 0) \ \textbf{reacts} - \textbf{to} \ \mathit{true} \\ \rangle^1\end{array}$$

In this definition, we introduce var, a logical table that allows us to refer to all variables in the system, referenced by the unique variable id. When selecting the variable a from the table, the statement above really selects its variable id, which serves as a reference to a specific entry in the table var. In this statement, for instance, the exposed variable a is non-deterministically selected from all exposed variables whose access control policies allow this agent access to read and write the exposed variable that the dummy variable a refers to. The latter is determined by applying the variable's access control policy to this agent's credentials. The set returned by this application can contain any combination of r and w, where the presence of the former indicates permission to read the variable, and the presence of the latter indicates permission to write the variable. After selecting the particular exposed variable to which a refers, the rule contains two assignments. The first assigns the value stored in a (i.e., $\mathsf{var}[a].\nu$) to the context variable c. The second assignment captures the fact that the context rule can also impact the environment, in this case by zeroing out the exposed variable used.

The power of the context-sensitive selection of exposed variables becomes apparent only when the restrictions within the context rules are used. Within the restrictions, the context rule can select exposed variables to be used based on the exposed variables' names, types, values, owning agent, or even based on properties of other variables belonging to the same or different agents. To simplify the specification of these restrictions, we introduce a few new pieces of notation. Referring to the system-wide table of variables (i.e., var) is cumbersome and confusing because the table is both virtual and distributed. For this reason, context rules refer directly to indexes in the table instead. Specifically, in this notation, we allow the variable id a to denote the value of the variable in var for entry a, i.e., $\mathsf{var}[a].\nu$. To access the other components of the variable (e.g., name), we abuse the notation slightly and allow $a.\eta$ to denote $\mathsf{var}[a].\eta$. Because a common operation in context-sensitive selection relies on selecting variables from the same program, we also introduce a shorthand for accessing a variable by the combination of name and program. To do this, when declaring dummy variables, a context rule can restrict both the names and relative owners of the variables. For example, the notation: $x \,!\, \mathsf{name}_1, y \,!\, \mathsf{name}_2 \ \mathit{in} \ p; z \,!\, \mathsf{name}_3 \ \mathit{in} \ q$ refers to three variables, one named name_1 and a second named name_2 that both belong to the

[1] The three-part notation $\langle \textbf{op} \ \mathit{quantified\_variable} : \mathit{range} :: \mathit{expression} \rangle$ used throughout the text is defined as follows: The variables from *quantified_variables* take on all possible values permitted by *range*. If *range* is missing, the first colon is omitted and the domain of the variables is restricted by context. Each such instantiation of the variables is substituted in *expression*, producing a multiset of values to which **op** is applied, yielding the value of the three-part expression. If no instantiation of the variables satisfies *range*, the value of the three-part expression is the identity element for **op**, e.g., *true* when **op** is $\forall$ or zero if **op** is "+" .

same agent whose agent_id can be referenced as p. The third variable, z, must be named name_3 and located in program q. q may or may not be the same as p, depending on further restrictions that might be specified. Additional variables can be listed in this declaration; they are grouped by program and separated by semicolons. If no combination of variables in the system satisfies the constraints, then the dummy variables are undefined, and the rule reduces to a skip.

As a simple example of a context rule, consider a program with a context variable called c that holds the value of an exposed variable with the name data and located on an agent at the same location as the reference. This context variable simply represents the context, and it does not change the data stored on the agent owning the exposed variable. To achieve this kind of behavior, the specification relies on the existence of the built-in exposed variable with the name location, locally referred to as λ. The context rule for the context variable c uses a single exposed variable that refers to the data that will be stored in c. In this example, we leave the rule unguarded, and it falls into the set of normal statements that are executed in a weakly-fair manner.

$$\begin{array}{ll} c\ \textbf{uses} & d\,!\,\mathsf{data}, l\,!\,\mathsf{location}\ \mathit{in}\ p \\ \quad\textbf{given} & l = \lambda \\ \quad\textbf{where} & c\ \textbf{becomes}\ d \end{array}$$

Formally, using the above notation is equivalent to the following expression:

$$\begin{array}{l} \langle d, l : (d, l) = (d', l').(\{r\} \subseteq \mathsf{var}[d'].\alpha(\mathsf{credentials}) \wedge \{r\} \subseteq \mathsf{var}[l'].\alpha(\mathit{credentials}) \wedge \\ \qquad\qquad \mathsf{var}[d'].\eta = \mathsf{data} \wedge \mathsf{var}[l'].\eta = \mathsf{location} \wedge \\ \qquad\qquad \mathsf{var}[d'].\pi = \mathsf{var}[l'].\pi \wedge \mathsf{var}[l'].\nu = \lambda.\nu) \\ \quad :: c := \mathsf{var}[d].\nu \\ \ \rangle \end{array}$$

Because the expression assigned to the context variable c is simply the value of the selected exposed variable, the most interesting portion of this expression is the non-deterministic selection of the exposed variables. The formal expression non-deterministically selects a variable to pull data from that satisfies a set of conditions. These conditions rely on the selection of a second exposed variable that stores the program's location. The first line of the non-deterministic selection checks the access control function for each of the variables to ensure that this agent is allowed read access given its credentials. The second line restricts the names of the two variables. The variable d being selected must be named data, according to the restrictions provided in the rule. The location variable is selected based on its name being location. The final line in the non-deterministic selection deals with the locations of the two variables. The first clause ensures that the two variables (d and l) are located in the same program. The second clause ensures that the agent that owns these two variables is at the same location as the agent defining the rule.

To show how these expressions can be used to facilitate modeling real-world context-aware interactions, we revisit the acquaintance list example from earlier in the section. More extensive examples will be discussed in Section 4.

In Fig. 2, we gave only a high level description of the context rules required to define an agent's acquaintance list. To define the membership qualifications exactly, the agent uses a context rule that adds qualifying agents to the context variable Q that stores the acquaintance list. In this particular case, assume that the program wants to restrict the acquaintance list members to other agents within some predefined range. This range is stored in a local variable whose local handle is referred to as *range*. The acquaintance list context variable can be defined using the following rule:

$$\begin{array}{ll} Q\ \textbf{uses} & l\,!\,\mathsf{location}\ \mathit{in}\ a \\ \quad\textbf{given} & |l - \lambda| \leq \mathit{range} \\ \quad\textbf{where} & Q\ \textbf{becomes}\ Q \cup \{a\} \\ \quad\textbf{reactive} & \end{array}$$

This expression uses the two handles *range* and λ to refer to local variables that store the maximum allowable range and the agent's current location, respectively. This statement adds agents that satisfy the membership requirements to the acquaintance list Q one at a time. Because it is a reactive statement that is enabled when an agent is within range, the rule ensures that the acquaintance list remains consistent with the state of the environment. As a portion of the reactive program that executes after each normal statement, this context rule reaches fixed-point when the acquaintance list contains all of the agents that satisfy the requirements for membership. An additional rule is required to eliminate agents that might still be in Q but are no longer in range:

$$\begin{array}{ll} Q\ \textbf{uses} & l\,!\,\mathsf{location}\ \mathit{in}\ a \\ \quad\textbf{given} & |l - \lambda| > \mathit{range} \\ \quad\textbf{where} & Q\ \textbf{becomes}\ Q - \{a\} \\ \quad\textbf{reactive} & \end{array}$$

Governing Universal Behaviors. Fig. 1 showed that the final portion of a Context UNITY system specification is a **Governance** section. It contains rules that capture behaviors that have universal impact across the system. These rules use the exposed variables available in programs throughout the system to affect other exposed variables in the system. The rules have a format similar to the definition of a program's local context rules except that they do not affect individual context variables:

$$\begin{array}{ll} \textbf{use} & \textit{quantified variables} \\ \textbf{where} & \textit{restrictions on quantified variables} \\ & \textit{expr}_1\ \textbf{impacts}\ \textit{exposed variable}_1 \\ & \textit{expr}_2\ \textbf{impacts}\ \textit{exposed variable}_2 \\ & \ldots \end{array}$$

As a simple example of governance, imagine a central controller that, each time its governance rule is selected, non-deterministically chooses an agent in the system and moves it, i.e., it models a random walk. This example assumes a one-dimensional space in which agents are located; essentially the agents can move along a line. Each agent's built-in location variable stores the agent's position on the line, and another variable named direction indicates which direction along the line the agent is moving. If the value of the direction variable is $+1$, the agent is moving in the positive direction along the line; if the value of the direction variable is -1, the agent is moving in the negative direction. We arbitrarily assume the physical space for movement is bounded by 0 on the low end and 25 on the upper end. The governance rule has the following form:

$$\begin{array}{ll} \textbf{use} & d\,!\,\mathsf{direction}, l\,!\,\mathsf{location}\ \mathit{in}\ p \\ \textbf{where} & l + d\ \textbf{impacts}\ l \\ & (\mathit{if}\ l + d = 25 \vee l - d = 0\ \mathit{then}\ -d\ \mathit{else}\ d)\ \textbf{impacts}\ d \end{array}$$

The non-deterministic selection clause chooses a d and l from the same program with the appropriate variable names. The first of the impact statements moves the agent in its current direction. The second impact statement switches the agent's direction if it has reached either boundary. The rules placed in the **Governance** section can be declared reactive, just as a local program's context rules are. The formal semantic definition of context rules in the **Governance** section differs slightly from the definition outlined above in that the governance rules need not account for the access control policies of the referenced exposed variables. This is due to the fact that the specified rules define system-wide interactions that are assumed, since they are provided by a controller, to be safe and allowed actions. As an example, the formal definition for the rule described above would be:

$$\begin{array}{l} \langle d, l : (d, l) = (d', l').(\mathsf{var}[l'].\eta = \mathsf{location} \wedge \mathsf{var}[d'].\eta = \mathsf{direction} \wedge \\ \qquad\qquad\qquad\qquad \mathsf{var}[l'].\pi = \mathsf{var}[d'].\pi) \\ \quad :: \ \mathsf{var}[l].\nu := \mathsf{var}[l].\nu + \mathsf{var}[d].\nu \\ \quad\ \ || \mathsf{var}[d].\nu := -\mathsf{var}[d].\nu\ \mathit{if}\ l + d = 25 \vee l + d = 0 \\ \quad \rangle \end{array}$$

Using the unique combination of independent programs, their context rules, and universal governance rules, Context UNITY possesses the ability to model a wide-variety of applications in the area of context-aware computing. We demonstrate this in Section 4 by providing snippets of Context UNITY systems required to model applications taken from the context-aware literature. First, in the next section, we briefly overview the proof logic associated with the Context UNITY model.

3 Proof Logic

Context UNITY has an associated proof logic largely inherited from Mobile UNITY [6], which in turn builds on the original UNITY proof logic [10]. Program properties are expressed using a small set of predicate relations whose validity can be derived directly from the program text, indirectly through translation of program text fragments into Mobile UNITY constructs, or from other properties through the application of inference rules. In all of these systems, the fundamental aspect of proving programs correct deals with the semantics of individual program statements. UNITY contains only standard conditional multiple assignment statements, while both Mobile UNITY and Context UNITY extend this model with reactive statements and transactions. Context UNITY also adds non-deterministic assignment statements. In all of these models, proving individual statements correct starts with the use of the *Hoare triple* [11].

For the normal statements in UNITY, a property such as:

$$\{p\}s\{q\} \textbf{ where } s \textbf{ in } P$$

refers to a standard conditional multiple assignment statement s exactly as it appears in the text of the program P. By contrast, in a Mobile UNITY or Context UNITY program, the presence of reactive statements requires us to use:

$$\{p\}s^*\{q\} \textbf{ where } s \in \mathcal{N}$$

where $\mathcal{N}$ denotes the normal statements of P while s^* denotes a normal statement s modified to reflect the extended behavior resulting from the execution of the reactive statements in the reactive program $\mathcal{R}$ consisting of all reactive statements in P. The following inference rule captures the proof obligations associated with verifying a Hoare triple in Context UNITY under the assumption that s is not a transaction:

$$\frac{\{p\}s\{H\}, H \mapsto (FP(\mathcal{R}) \wedge q) \text{ in } \mathcal{R}}{\{p\}s^*\{q\}}$$

The first component of the hypothesis states that, when executed in a state satisfying p, the statement s establishes the intermediate postcondition H. This postcondition serves as a precondition of the reactive program $\mathcal{R}$, that, when executed to fixed-point, establishes the final postcondition q. The "in $\mathcal{R}$" must be added because the proof of termination is to be carried out from the text of the reactive statements, ignoring other statements in the system. This can be accomplished with a variety of standard UNITY techniques. It is required that the predicate H leads to a fixed-point and q in the reactive program $\mathcal{R}$. This proof obligation (i.e., $H \mapsto (FP(\mathcal{R}) \wedge q)$ in $\mathcal{R}$) can be proven with standard techniques because $\mathcal{R}$ is treated as a standard UNITY program.

For transactions of the form $\langle s_1; s_2; \ldots; s_n\rangle$ we can use the following inference rule before application of the one above:

$$\frac{\{a\}\langle s_1; s_2; \ldots s_{n-1}\rangle^*\{c\}, \{c\}s_n^*\{b\}}{\{a\}\langle s_1; s_2; \ldots s_n\rangle\{b\}}$$

where c may be guessed at or derived from b as appropriate. This represents sequential composition of a reactively-augmented prefix of the transaction with its last sub-action. This rule can be used recursively until we have reduced the transaction to a single sub-action. Then we can apply the more complex rule above to each statement. This rule may seem complicated, but it represents standard axiomatic reasoning for ordinary sequential programs, where each sub-statement is a predicate transformer that is functionally composed with others.

Finally, Context UNITY introduces the notion of non-deterministic assignment to the Mobile UNITY proof logic. The proof obligation of these non-deterministic assignments differs slightly from that of the standard assignment statements. Given the property $\{p\}s\{r\}$ in UNITY, if the statement s is a non-deterministic assignment statement of the form $x := x'.Q(x')$, then the inference rule describing the associated proof obligation for the statement s has the form:

$$\frac{\{p \wedge \exists x' :: Q(x')\}s\{\forall x' : Q(x') :: r\}}{\{p\}s\{r\}}$$

Special care must be taken to translate Context UNITY context rules from both the local program **context** sections and the **Governance** section to standard notation (i.e., to the appropriate normal or reactive statements) before applying the proof logic outlined here. Once translated as described in the previous section, proof of the system can be accomplished directly by applying the rules outlined above.

To prove more sophisticated properties, UNITY-based models use predicate relations. Basic safety is expressed using the **unless** relation. For two state predicates p and q, the expression p **unless** q means that, for any state satisfying p and not q, the next state in the execution must satisfy either p or q. There is no requirement for the program to reach a state that satisfies q, i.e., p may hold forever. Progress is expressed using the **ensures** relation. The relation p **ensures** q means that for any state satisfying p and not q, the next state must satisfy p or q. In addition, there is some statement in the program that guarantees the establishment of q if executed in a state satisfying p and not q. Note that the **ensures** relation is not itself a pure liveness property but is a conjunction of a safety and a liveness property; the safety part of the **ensures** relation can be expressed as an **unless** property. In UNITY, these predicate relations are defined by:

$$p \textbf{ unless } q \equiv \langle \forall s : s \textbf{ in } P :: \{p \wedge \neg q\}s\{p \vee q\}\rangle$$

$$p \textbf{ ensures } q \equiv (p \textbf{ unless } q) \wedge \langle \exists s : s \textbf{ in } P :: \{p \wedge \neg q\}s\{q\}\rangle$$

where s is a statement in the program P. Mobile UNITY and Context UNITY use the same definitions since all distinctions are captured in the verification of the Hoare triple. Additional relations may be derived to express other safety (e.g., **invariant** and **stable**) and liveness (e.g., **leads-to**) properties.

4 Patterns of Context-Awareness

Much published research acknowledges the need for applications that rapidly adapt to changes in resource availability and the operational environment. As a result, a number of researchers sought to provide context-aware software systems designed to function in a variety of operating scenarios. These systems vary in their approaches to managing context; models that underlie context-aware systems range from a simple client-server model in which servers provide context information directly to clients, to sophisticated tuple space coordination models in which the details of communicating context information is transparent to the application. In this section, we examine a representative set of context-aware systems found in the literature, abstract their key features, and suggest ways to model them in Context UNITY.

4.1 Simple Context Interactions

Initial work in context-aware computing resulted in the development of applications that use relatively simple context definitions. Such systems often separate concerns related to providing and using context. Many systems consist of *kiosks*, entities which provide context information to *visitors*, which use context and state information to adapt their behavior.

Applications exhibiting the characteristics of the simple kiosk-visitor interaction pattern include context-aware office applications such as Active Badge [12] and PARCTab [13]. In these systems, personnel carry devices that periodically communicate a unique identifier via a signal to fixed sensors, allowing the location of the carrier to be known. An application uses the location information to adapt the office environment accordingly in response to the changing location of the carrier, e.g., by forwarding phone calls to the appropriate office or changing the applications available on a workstation. Another type of context-aware applications that use simple context interactions relate to the development of tour guides, e.g., Cyberguide [14] and GUIDE [15]. In these applications, tourists carry mobile devices equipped with context-aware tour guide software. As a tourist moves about in a guide-friendly area, his display is updated according to locally stored preferences combined with context information provided by stationary access points located at points of interest.

In all of the context-aware applications described above, a particular type of entity provides context information and another type reads and uses the provided information. Generally, one of the parties is stationary, while the other is mobile. We can readily capture this style of interaction in Context UNITY. Agents providing context information to other agents in the Context UNITY system do so through the use of *exposed* variables. Agents obtain the provided context information through the use of *context* variables, the values of which are defined by values of selected exposed variables of context-providing agents.

Fig. 4 illustrates the interaction between a visitor and kiosks in a simple museum guide system. In this system, each stationary museum kiosk provides information about an exhibit at its location using an exposed variable. A kiosk in the southeast corner of the museum gives information about a painting through its exposed variable e named "painting" with a textual description of the painting as the variable's value. The kiosks in the northeast and northwest corners of the museum each provide information about a certain sculpture by naming its exposed variable e "sculpture," and assigning to the variable a short textual description of the work of art at that location. As a particular visitor moves around the room, his context variable, c, defined to contain a co-located sculpture exhibit, changes in response to the available context. If new context information about a sculpture is available, the visitor's display is updated to show the information. The figure depicts what happens when a visitor walks around the museum. The initial position of the visitor agent is depicted by the dashed box labeled "Agent." As the visitor moves around the museum in the path indicated by the dotted arrow, the context variable c is updated. Specifically, when the visitor reaches the northeast corner of the museum, the context variable c is updated to contain information about the sculpture at that location. Such an application can be specified in the Context UNITY notation, as shown below.

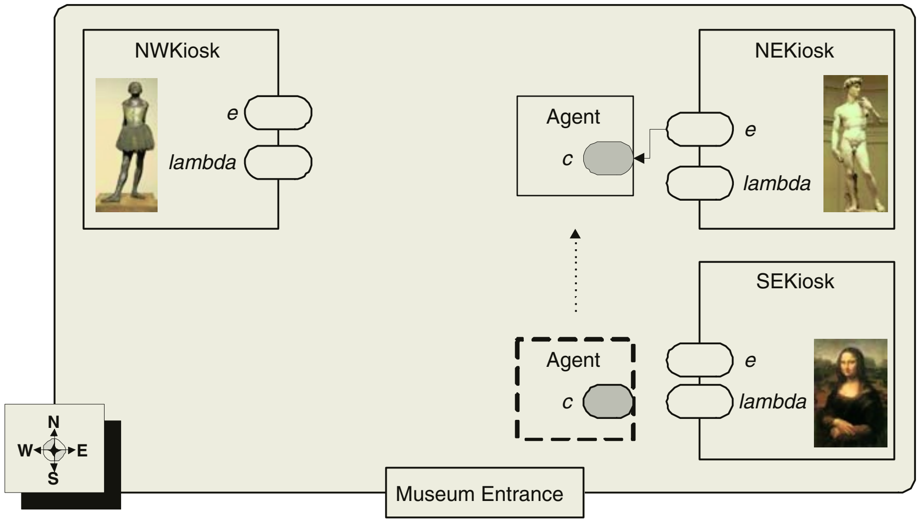


Fig. 4. A simple guide system in Context UNITY

For brevity, we show only the most interesting aspect of the system specification, which is a visitor's context rule:

$$\begin{array}{ll} c\ \textbf{uses} & e\,!\,\mathsf{sculpture}, l\,!\,\mathsf{location}\ \mathit{in}\ p \\ \quad\textbf{given} & l = \lambda \\ \quad\textbf{where} & c\ \textbf{becomes}\ e \end{array}$$

More complex patterns of interaction are frequently utilized in the development of context-aware systems. In some systems, for instance, kiosks provide context information to a stationary context manager, and the context manager communicates directly with visitors to adapt their behavior accordingly given the current conditions of the environment. An instance of this pattern of interaction is found in the Gaia operating system [16], which manages *active spaces*. An active space is a physical location in which the physical and logical resources present can be adapted in response to changes in the environment. A typical interaction in an active space is as follows: a user enters the active space and registers with the context manager, which uses information about the user and the environment to perform appropriate actions, e.g., turn on a projector and load the user's presentation. Such a system can be modeled in Context UNITY similarly to those systems described above that exhibit simple context interactions: users are providing context information to the context manager through the use of exposed variables, and the context manager uses context variables to obtain context information and react accordingly.

4.2 Security-Constrained Context Interactions

Security is a major concern in the development of all modern software systems, including those supporting context-awareness. In several systems, multi-level security mechanisms are provided through the use of *domains*. A domain provides a level of security and isolates the available resources according to the level of security offered. Agents authorized to operate within that domain have the ability to act upon all resources within a domain, and a domain may have an authorizing authority that grants and revokes entering and exiting agents' access rights. Examples of systems exhibiting such characteristics include the Gaia file system [16] and the multi-level access control proposed by Wickramasuriya and Venkatasubramanian [17].

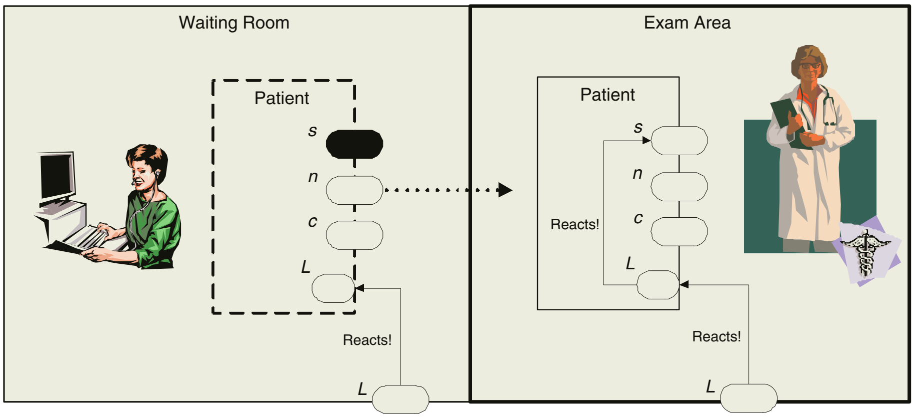


Fig. 5. An example security-constrained context-aware application in Context UNITY. In the waiting room domain, which offers a low level of security in its exposed variable L, the patient's sensitive information about symptoms is protected from inclusion in the domain by the symptom variable's access control function. The shading on the oval labeled s indicates that the symptom variable is not accessible to anyone in the environment. As the patient moves to the exam area domain offering high level security, the patient's domain security level is updated immediately, as indicated by the arrow labeled "reacts." As a result of the changed security level, a second reaction is triggered whose effect is to alter the access control function of the symptom variable s to allow the value to be available to those in the exam area domain.

Fig. 5 illustrates an example use of such an interaction style. In the example, a patient at a doctor's office must provide information about himself in order to receive treatment. Some of the information provided is fairly public knowledge and can be viewed by the receptionist and other patients, e.g., name and contact information. Other information is highly sensitive and personal, e.g., health history, and should only be shared with a doctor. To facilitate this kind of interaction, the doctor's office is divided into two areas that provide different levels of privacy: the waiting room and the exam area. The waiting room is a public space (low-security), since the receptionist and other patients in the waiting room can view the information provided therein. The exam area is private (high-security), since only the patient and doctor can view the information.

To describe such applications in Context UNITY, domains could reveal their security level using an exposed variable L named "security level." Each patient agent uses a context rule for its context variable L to discover the level of security offered by the domain in which it is located. Because the definition is built to be strongly consistent using a reactive statement, the agent's perception of the security level offered by its current domain is guaranteed to be accurate and up to date. Each patient provides his name, contact information, and symptoms through the use of exposed variables n, c, and s. A patient controls how his information is made available through the use of each variable's access control function. This access control function can be changed during the execution of the program to reflect the agent's changing data protection needs. Using a reaction, it is possible to ensure that the access control function is immediately changed to reflect a change in the security level as soon as a new domain (and hence, a new level of security) is entered.

4.3 Tailored Context Definitions

Often, the amount of context information available to a context-aware agent grows large and unmanageable. To avoid presenting an agent with an overwhelming amount of context in such a scenario, it is desirable to limit the amount of context information that the agent "sees" based on properties of its environment. An example of a context-aware system that does just this is EgoSpaces [3], a middleware for use in ad hoc networks. At the heart of EgoSpaces is the *view* concept, which restricts an agent's context according to the agent's individualized specification. A view consists of constraints on network properties, the agents from which context is obtained, and the hosts on which such agents reside. These constraints are used to filter out unwanted items in the operational environment and results in presenting the agent with a context (view of the world) tailored to its particular needs.

In a general sense, systems such as EgoSpaces consist of possibly mobile agents that are both providers and users of context, and a context management strategy that is performed on a per-agent basis. An individualized context is managed on behalf of each agent by matching items from the entire operational environment against the restrictions provided in the view definition, and presenting the result to the agent as its context. Such a system can be readily expressed in Context UNITY. To act as a context provider, an agent generates pieces of context information and places them in an exposed variable, a tuple space, in the case of EgoSpaces, i.e., a data repository consisting of tuples that the agent wishes to contribute as context. An agent provides information about itself and properties about the host on which it resides in exposed variables named "agent profile" and "host profile," respectively. They allow other agents to filter the operational environment according to the host and agent constraints in their view definitions. To act as a context user, we model an agent's view using a rule for a context variable v named "view." The value of v is defined to be the set of all tuples present in exposed tuple space variables of other reachable agents for which the exposed agent profile properties, exposed host profile properties, and exposed network properties of hosts match the reference agent's constraints. An example context rule that establishes a view v for an agent with id i to "see" can be described as follows:

$$\begin{array}{ll}
v\ \textbf{uses} & \mathit{lts}\ !\ \mathsf{tuple\_space},\ a\ !\ \mathsf{agent\_profile},\ h\ !\ \mathsf{host\_profile}\ \mathit{in}\ i \\
\quad\textbf{given} & \mathit{reachable}(i) \wedge \mathit{eligibleAgent}(a) \wedge \mathit{eligibleHost}(h) \\
\quad\textbf{where} & v\ \textbf{becomes}\ v - (v \uparrow i)\ \cup\ \mathit{lts} \\
\quad\textbf{reactive} &
\end{array}$$

The function *reachable* encapsulates the network constraints that establish whether an agent should or should not be considered based on network topology data. The notation $v \uparrow i$ indicates a projection over the set v that contains tuples owned by the agent i. It is possible to obtain such a projection since we assume that each generated tuple has a field which identifies the owner of the tuple using the generating agent's unique id. In order for an agent to perform changes to the view v and have them propagate to the correct tuple space *lts* additional context rules are needed.

4.4 Uniform Context Definition

Coordination models offer a high degree of decoupling, an important design characteristic of context-aware systems. In many distributed computing environments, tuple spaces are permanently attached to agents or hosts. In some models, these pieces merge together to logically form a single shared tuple space in a manner that takes into consideration the connectivity among agents or hosts. An agent interacts with other agents by employing content-based retrieval (`rd(pattern)` and `in(pattern)`), and by generating tuples (`out(tuple)`). Often, the traditional operations are augmented with reactions that extend their effects to include arbitrary atomic state transitions. Systems borne out of such a tuple space coordination paradigm can be considered context-aware; an agent's context is managed by the tuple space system in the form of tuples in a logically shared tuple space.

Examples of such context-aware systems are TSpaces [18], JavaSpaces [19], MARS [20], and LIME [21]. A common characteristic of all these systems is the fact that agents that enter in a sharing relation have the same definition of context, i.e., the context rules are uniform and universally applied. Among the systems we cite here, LIME is the most general, as it incorporates both physical mobility of hosts and logical mobility of agents, and provides tuple space sharing in the most extreme of network environments – the ad hoc network. In LIME, agents are units of execution, mobility, and data storage, while hosts are simply containers of agents. Hosts may be mobile, and agents can migrate from host to host. Agents may be associated with several local tuple spaces, distinguished by name. Since it is a passive entity, a host has no tuple space. A LIME agent's relevant context is determined by the logically merged contents of identically named tuple spaces held by mutually reachable agents.

To capture the essential features of context-aware systems having the characteristics described above in Context UNITY, it suffices to endow an agent with one exposed variable named localTS that offers its local tuple space for sharing and a second exposed variable named sharedTS that should provide access to all the tuples making up the current context. The value of the latter is the union of tuples contained in exposed local tuple space variables belonging to connected agents. Connectivity can be defined based on various properties of the network, e.g., network hops, physical distance, etc. In MARS, only agents residing on the same host are *connected*. In LIME, agents are *connected* when residing on the same host or on physically connected hosts.

A final and important point to note about the modeling of such systems is that since the shared tuple space definition is uniform across all agents, we can capture it in the **Governance** section of a Context UNITY system. While it is possible to define an agent's context locally in its program description, using the **Governance** section highlights the fact that connected agents share a symmetric context. In addition, it is more economical for a programmer to write a single context definition since it applies to the entire system. The resulting context rule included in the **Governance** section is as follows:

use ts_c ! sharedTS *in* a; ts_l ! localTS *in* b
given *connected*(a, b)
where $ts_c \;-\; (ts_c \uparrow b) \;\cup\; ts_l$ **impacts** ts_c
reactive

The result of this context rule is a tuple space shared among connected agents.

This brings to an end our discussion on how Context UNITY relates to some of the existing models of context-awareness. The most striking observation about this informal evaluation of the model is the simplicity exhibited by each of the context rules that were generated in this section.

5 Conclusions

The formulation of the Context UNITY model is a case study designed to help us gain a better understanding of the essential features of the context-aware computing paradigm. A key feature of the model is the delicate balance it achieves between placing no intrinsic limits on what the context can be while empowering the individual agent with the ability to precisely control the context definition. Linguistically the distinction is captured by the notions of operational environment and context, expansive with respect to potential and specific with respect to relevance. In the model, the two concepts have direct representations in terms of exposed and context variables. The other fundamental characteristic of the model is rooted in the systematic application of software engineering methodological principles to the specifics of context-aware computing. The functionality of the application code is separated from the definition of context. This decoupling is fundamental in a setting where adaptability is important – a program design cannot anticipate the details of the various operational environments the program will encounter throughout its life time. The model enables this decoupling through the introduction of context rules that exploit existential quantification and non-determinism in order to accommodate the unknown and unexpected. Context UNITY explicitly captures the essential characteristics of context-awareness, as we experienced then in our work and observed them in that of others. Moreover, the defining traits of many existing models appear to have simple and straightforward representations in Context UNITY, at least at an abstract level. While we acknowledge that further refinements and evaluation of the model are needed, all indications to date suggest that the essential features of context-aware computing are indeed present in the model.

Acknowledgements

This research was supported in part by the Office of Naval Research under ONR MURI research contract N00014-02-1-0715. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

1. Salber, D., Dey, A., Abowd, G.: The Context Toolkit: Aiding the development of context-enabled applications. In: Proceedings of CHI'99. (1999) 434–441
2. Hong, J., Landay, J.: An infrastructure approach to context-aware computing. Human Computer Interaction **16** (2001)
3. Julien, C., Roman, G.C.: Egocentric context-aware programming in ad hoc mobile environments. In: Proceedings of the 10^{th} International Symposium on the Foundations of Software Engineering. (2002) 21–30
4. Schilit, B., Adams, N., Want, R.: Context-aware computing applications. In: IEEE Workshop on Mobile Computing Systems and Applications. (1994)
5. Julien, C., Payton, J., Roman, G.C.: Reasoning about context-awareness in the presence of mobility. In: Proceedings of the 2^{nd} International Workshop on Foundations of Coolrdinations Languages and Software Architectures. (2003)
6. Roman, G.C., McCann, P.J.: A notation and logic for mobile computing. Formal Methods in System Design **20** (2002) 47–68
7. McCann, P.J., Roman, G.C.: Compositional programming abstractions for mobile computing. IEEE Transactions on Software Engineering **24** (1998) 97–110
8. Fok, C.L., Roman, G.C., Hackmann, G.: A lightweight coordination middleware for mobile computing. In: Proceedings of the 6^{th} International Conference on Coordination Models and Languages. (2004) (to appear).
9. Back, R.J.R., Sere, K.: Stepwise refinement of parallel algorithms. Science of Computer Programming **13** (1990) 133–180
10. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley, NY, USA (1988)
11. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12** (1969) 576–580, 583
12. Harter, A., Hopper, A.: A distributed location system for the active office. IEEE Networks **8** (1994) 62–70

13. Want, R., et al.: An overview of the PARCTab ubiquitous computing environment. IEEE Personal Communications **2** (1995) 28–33
14. Abowd, G., Atkeson, C., Hong, J., Long, S., Kooper, R., Pinkerton, M.: Cyberguide: A mobile context-aware tour guide. ACM Wireless Networks **3** (1997) 421–433
15. Cheverst, K., Davies, N., Mitchell, K., Friday, A., Efstratiou, C.: Experiences of developing and deploying a context-aware tourist guide: The GUIDE project. In: Proceedings of MobiCom, ACM Press (2000) 20–31
16. Roman, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R.: Gaia: A middleware infrastructure to enable active spaces. IEEE Pervasive Computing (2002) 74–83
17. Wickramasuriya, J., Venkatasubramanian, N.: A middleware approach to access control for mobile concurrent objects. In: Proceedings of the International Symposium on Distributed Objects and Applications. (2002)
18. IBM: T Spaces. http://www.almaden.ibm.com/cs/TSpaces/ (2001)
19. Sun: Javaspaces. http://www.sun.com/jini/specs/jini1.1html/js-title.html (2001)
20. Cabri, G., Leonardi, L., Zambonelli, F.: MARS: A programmable coordination architecture for mobile agents. Internet Computing **4** (2000) 26–35
21. Murphy, A.L., Picco, G.P., Roman, G.C.: Lime: A middleware for physical and logical mobility. In: Proceedings of the 21st International Conference on Distributed Computing Systems. (2001) 524–533

Consistent Adaptation and Evolution of Class Diagrams during Refinement

Alexander Egyed

Teknowledge Corporation, 4640 Admiralty Way, Suite 1010
Marina Del Rey, CA 90034, USA
aegyed@ieee.org

Abstract. Software models are key in separating and solving independent development concerns. However, there is still a gap on how to transition design information among these separate, but related models during development and maintenance. This paper addresses the problem on how to maintain the consistency of UML class diagrams during various levels of refinement. We present a new approach to automated consistency checking called *ViewIntegra*. Our approach separates consistency checking into transformation and comparison. It uses transformation to translate model elements to simplify their subsequent comparison. Transformation-based consistency checking, in the manner we use it, is new since we use transformation to bridge the gap between software models. No intermediate models or model checkers are required; developers need only be familiar with the models they design with and none other. The separation of transformation and comparison makes our approach to consistency checking more transparent. It also makes our approach useful for both propagating design changes among models and validating consistency. This gives developers added flexibility in deciding when to re-generate a model from scratch or when to resolve its inconsistencies. Although this paper emphasizes the adaptation and evaluation of class diagrams, we found our technique to be equally useful on other models. Our approach is tool supported.

1 Introduction

In the past decades, numerous software models were created to support software development at large. Models usually break up software development into smaller, more comprehensible pieces utilizing a divide and conquer strategy. The major drawback of models is that development concerns cannot truly be investigated all by themselves since they depend on one another. If a set of issues about a system is investigated, each through its own models, then the validity of a solution derived from those models requires that commonalities (redundancies) between them are recognized and maintained in a consistent fashion. Maintaining consistency between models is a non-trivial problem. It is expensive and labor intensive despite the vast number of past and existing research contributions.

In this work, we introduce a transformation-based approach to consistency checking called *ViewIntegra*. This paper describes our approach to automated consistency checking and shows how to translate models to simplify their comparison. The effect is that redundant information from one model is re-interpreted in the context and language of another model followed by a simple, one-to-one comparison to detect differences. We limit the discussion in this paper to the abstraction and refinement of UML-like class diagrams [24]. We believe our approach to be equally applicable to other kinds of models.

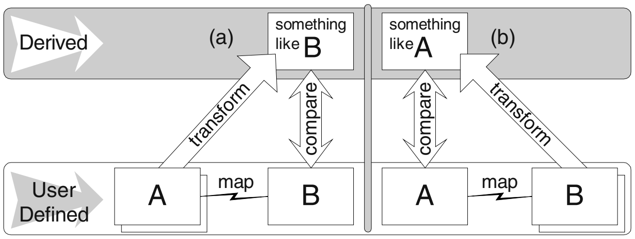


Fig. 1. View transformation and mapping to complement view comparison

Figure 1 depicts the principle of transformation-based consistency checking. In order to compare the two (user-defined[1]) models A and B (e.g., high-level model and low-level model), we transform one of them into 'something like the other' so that the one becomes easier comparable to the other. For example, our approach transforms the low-level class diagram into a form that makes the results directly comparable with the high-level diagram. As part of our collaboration with Rational Corporation we created such an transformation technique [9] and, in this paper, we will demonstrate how its results can be used for direct, one-to-one comparison to detect inconsistencies.

Our approach separates the propagation of design information (transformation) from the comparing of design information (consistency checking). It follows that transformation may be used independently from comparison for change propagation. For example, the above mentioned transformation technique can be used during reverse engineering to generate a high-level class diagram from a lower-level one; or the transformation technique can be used during consistency checking to suggest its transformation results as an option for resolving inconsistencies.

Traceability among model elements is needed to guide transformation and comparison. We found that it is typically very hard to generate traceability information in detail although developers are capable of approximating it [6]. This would be a problem for any consistency checking approach but our approach can alleviate this problem significantly. This paper will thus also demonstrate how our approach behaves with a partial lack of traceability information.

[1] User-defined views are diagrams that are created by humans (e.g., Fig. 2.). Derived views (interpretations) are diagrams that are automatically generated via *Transformation*.

Because our approach separates transformation from comparison during consistency checking, it also benefits from reuse of previously transformed, unchanged design information. This greatly improves performance because subsequent comparisons require partial re-transformations only. Another benefit is that consistency rules are very generic and simple since they have to compare alike model elements only.

We evaluated our transformation-based approach to consistency checking on various types of heterogonous models like class diagrams, state chart diagrams, sequence diagrams [8], and the C2SADEL architecture description language [10]. Furthermore, we validated the usefulness of our approach (in terms of its scope), its correctness (e.g., true errors, false errors), and scalability via a series of experiments using third-party models [1,2] and in-house developed models. Our tool support, called UML/Analyzer, fully implements transformation-based consistency checking for class and C2SADEL diagrams. The other diagrams are partially supported only.

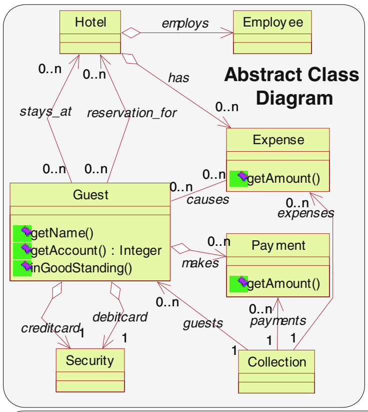
The remainder of this paper is organized as follows: Section 2 introduces an example and discusses abstraction and refinement problems in context of two class diagrams depicted there. Section 3 will highlight consistency checking without transformation and discusses in what cases it is effective and where it fails. Section 4 introduces a transformation technique for abstracting class diagrams and discusses how it improves the scope of detectable inconsistencies. Section 4 discusses how our transformation and consistency checking methods are also able to interpret incomplete and ambiguous model information. Section 5 discusses issues like scope, accuracy, and scalability in more detail and Section 6 summarizes the relevance of our work with respect to related work.

2 Example

Figure 2 depicts two class diagrams of a *Hotel Management System* (HMS) at two levels of abstraction. The top diagram is the higher-level class diagram with classes like *Guest* and *Hotel*, and relationships like "a guest may either stay at a hotel or may have reservations for it." This diagram further states that a *Hotel* may have *Employees* and that there are *Expense* and *Payment* transactions associated with guests (and hotels). It is also indicated that a *Guest* requires a *Security* deposit. The diagram uses three types of relationships to indicate uni-directional, bi-directional, and part-of dependencies (see UML definition [24]). For instance, the relationship with the diamond head indicates aggregation (part-of) implying that, say, *Security* is a part of *Guest*. Additionally, the diagram lists a few methods that are associated with classes. For instance, the class *Expense* has one method called *getAmount()*.

The bottom part of Figure 2 depicts a refinement of the left side. Basic entities like *Guest* or *Expense* are still present although named slightly differently[2] and additional classes were introduced. For instance, the lower-level diagram makes use of new classes like *Reservation* or *Check* to refine or extend the higher-level diagram. The

[2] It must be noted that we use a disjoint set of class names in order to avoid naming confusions throughout this paper. Duplicate names are allowed as part of separate name spaces.



High-level Diagram	Low-level Diagram
Guest	*GuestEntity*
Hotel	*HotelEntity, Room, HotelCollection*
Security	*PaymentSecurity*
Expense	*ExpenseTransaction* and *Room Fee*
Payment	*PaymentTransaction*
Collection	*GuestCollection*, and *HotelCollection*

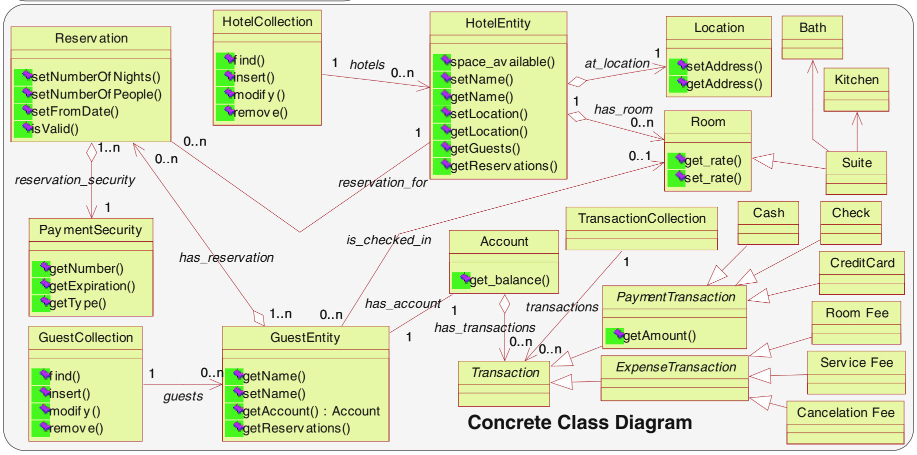


Fig. 2. High-level and low-level class diagrams of HMS and mapping table

refined class diagram uses the same types of relationships as the high-level one plus generalization relationships (triangle head) to indicate inheritance. The low-level diagram also describes methods associated with classes more extensively.

Both diagrams in Figure 2 separately describe the structure of the HMS system. Obviously, there must be some commonality between them (i.e., redundancy). For instance, the high-level class *Guest* is equivalent to the low-level class *GuestEntity* and the high-level relationship "*has*" (between *Guest* and *Expense*) is equivalent to the combined relationships with the classes *Transaction* and *Account* between them. The table in Figure 2 describes several cases of one-to-many mappings such as Hotel maps to *HotelEntity*, *Room*, *HotelCollection*, one case of a many-to-many mapping (there are several choices of how *reservation_for* and *stays_at* map to the low-level diagram), one case of a many-to-one mapping (*HotelCollection* is assigned to *Hotel* and *Collection*), and many cases of no mappings altogether (e.g., *Employee* in the high-level diagram or *Account* in the low-level diagram).

Knowledge on how model elements in separate diagrams relate to one another is commonly referred to as traceability (or mapping) [13]. Traceability is usually generated manually [16] (either by using common names or maintaining trace tables as depicted in Figure 2) but there exists some automation [6]. Like other consistency checking approaches, our approach requires some traceability knowledge but a discussion on how to derive it is out of the scope of this paper.

3 Simple Consistency Checking

Current consistency checking approaches detect inconsistencies by transforming models into some third-party language followed by constraint-based reasoning (often model checking) in context of that language. For instance, consistency checking approaches like JViews (MViews) [14], ViewPoints [15] or VisualSpecs [4] read diagrams, translate them into common (and usually formal) representation schemes, and validate inconsistency rules against them. These approaches have shown powerful results; they are precise and computationally efficient. But there are also unresolved side effects that one could well argue to be outside the scope of consistency checking but that are related and significant:

(1) Lack of Change Propagation: Existing approaches solve the problem of detecting inconsistencies very well but they lack support for the subsequent, necessary adaptation of models once inconsistencies are found.
(2) Lack of Traceability: Existing approaches require complete traceability to guide consistency checking. Generating traceability is a hard, error-prone activity with a potentially little life span.

4 Transformation-Based Consistency Checking

We describe our approach to automated consistency checking next and also discuss how it is able to handle above side effects. Our approach, called *ViewIntegra* [8], exploits the redundancy between models: for instance, the high-level diagram in Figure 2 contains information about the HMS system that is also captured in the low-level diagram. This redundant information can be seen as a constraint. Our approach uses transformation to translate redundant model information to simplify their comparison. The effect is that redundant information from one model is re-interpreted in the context and language of another model followed by a simple, one-to-one comparison to detect differences (effectively enforcing the constraint).

Abstraction Implementing Transformation

In the course of evaluating nine types of software models [8] (class, object, sequence, and state chart diagrams, their abstractions and C2SADEL) we identified the need for four transformation types called *Abstraction, Generalization*, *Structuralization*, and

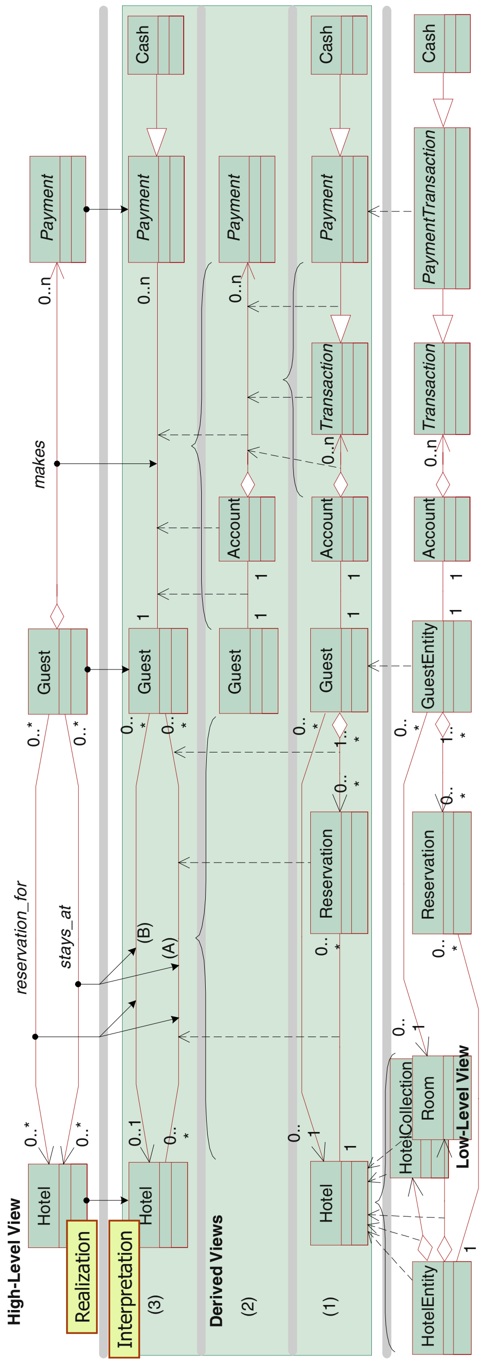


Fig. 3. Abstraction applied on part of HMS showing high-level, low-level, and derived modeling information

Translation. This paper focuses on inconsistencies during refinement and thus only needs *Abstraction*. See [8] for a discussion on the other types.

Abstraction deals with the simplification of information by removing details not necessary on a higher level. In [7], we identified two types of abstractions called *compositional abstraction* and *relational abstraction*. Compositional abstraction is probably the more intuitive abstraction type since it closely resembles hierarchical decomposition of systems. For instance, in UML, a tree-like hierarchy of classes can be built using a feature of classes that allows them to contain other classes. Thus, a class can be subdivided into other classes. In relational abstraction it is the relations (arrows) and not the classes that serve as vehicles for abstraction. Relations (with classes) can be collapsed into more abstract relations. Relational abstraction is needed since it is frequently not possible to maintain a strict tree-hierarchy of classes. Our abstraction technique has been published previously; we will provide a brief summary here only. For a more detailed discussion, please refer to [7,9].

In order to abstract the low-level diagram in Figure 2, we have to apply both abstraction types. Figure 3 shows a partial view of Figure 2 depicting, in the top layer, the high-level classes *Hotel*, *Guest*, and *Payment* and, in the bottom layer, their low-level counterparts *HotelEntity*, *HotelCollection*, *Room*, *GuestEntity*, and *PaymentTransaction*. The bottom layer also depicts relationships among the low-level classes.

Table 1. Excerpt of abstraction rules for classes [9]

1) Class x Association x Class x AggregationRight x Class equals Association 100
2) Class x AggregationLeft x Class x AssociationLeft x Class equals AssociationLeft 100
3) Class x Association x Class x AggregationLeft x Class equals Association 90
4) Class x AggregationLeft x Class x GeneralizationLeft x Class equals AggregationLeft 100
5) Class x GeneralizationLeft x Class x GeneralizationLeft x Class equals GeneralizationLeft 100
6) Class x DependencyRight x Class x AggregationRight x Class equals DependencyRight 100
7) Class x AssociationRight x Class x GeneralizationRight x Class equals AssociationRight 70
8) Class x Aggregation x Class equals Class 100

The first abstraction step is to use compositional abstraction to group low-level classes that belong to single high-level classes. For instance, the low-level classes *HotelEntity*, *HotelCollection*, and *Room* are all part of the same high-level class *Hotel* (recall traceability in Figure 2). The grouped class, called *Hotel*, is depicted in the first (1) derived view in Figure 3. Besides grouping the three low-level classes, the abstraction method also replicated the inter-dependencies of those three classes for the derived, high-level class. It can be seen that the derived class *Hotel* now has relationships to *Reservation* and *Guest* that were taken from *HotelEntity* and *Room* respectively. Also note that the single low-level classes *GuestEntity* and *Payment-Transaction* were grouped into the more high-level, derived classes *Guest* and *Payment*. They also took all inter-relationships from their low-level counterparts. Compositional abstraction simplified the low-level class diagram somewhat but it is still not possible to compare it directly to the high-level diagram. To simplify comparison, helper classes such as *Reservation*, *Account*, and *Transaction* need to be eliminated since they obstruct our understanding on the high-level relationships between classes such as *Hotel* and *Guest*. The problem is that those classes were not assigned to any high-level classes and thus could not be eliminated via compositional abstraction.

The second abstraction step is to group low-level relationships into single high-level relationships. For instance, the low-level relationship path going from *Guest* via *Reservation* to *Hotel* in Figure 3 (bottom) may have some abstract meaning. This meaning can be approximated through simpler, higher-level model elements. In particular, this example shows an aggregation relationship between the classes *Reservation* and *Guest* (diamond head) indicating that *Reservation* is a part of *Guest*. The example also shows a uni-directional association relationship from *Hotel* to *Reservation* indicating that *Reservation* can access methods and attributes of *Hotel* but not vice versa. What the diagram does not depict is the (more high-level) relationship between *Guest* and *Hotel*. Semantically, the fact that *Reservation* is part of a *Guest* implies that the class *Reservation* is conceptually *within* the class *Guest*. Therefore, if *Reservation* can access *Hotel*, *Guest* is also able to access *Hotel*. It follows that *Guest* relates to *Hotel* in the same manner as *Reservation* relates to *Hotel* making it possible for us to replace *Reservation* and its relationships with a single *association* originating in *Guest* and terminating in *Hotel* (third derived view in Figure 3).

In the course of inspecting numerous UML-type class diagrams, we identified over 120 class abstraction rules [9]. Table 1 shows a small sample of these abstraction rules as needed in this paper. Abstraction rules have a *given* part (left of "equals") and an *implies* part (right of "equals"). Rules 1 and 8 correspond to the two rules we discussed so far. Since the directionality of relationships is very important, the rules in Table 1 use the convention of extending the relationship type name with “Right” or “Left” to indicate the directions of their arrowheads. Furthermore, the number at the end of the rule indicates its reliability. Since rules are based on semantic interpretations, those rules may not always be valid. We use reliability numbers as a form of priority setting to distinguish more reliable rules from less reliable ones. Priorities are applied when deciding what rules to use when. We will later discover that those reliability numbers are also very helpful in determining false errors. It must be noted that rules may be applied in varying orders and they may also be applied recursively. Through recursion is it possible to eliminate multiple helper classes as in the case of the path between *PaymentTransaction* and *GuestEntity* (see second (2) and third (3) derived views in Figure 3).

Compositional and relational abstraction must be applied to the entire low-level diagram (Figure 2). A part of the resulting abstraction is depicted in the second row in Figure 3 (third (3) view). We refer to this result as the *interpretation* of the low-level diagram. This interpretation must now be compared to the high-level diagram.

Comparison

The abstraction technique presented above satisfies our criteria of a good transformation method because it transforms a given low-level class diagram into ‘something like’ the high-level class diagram. Obviously, consistency checking is greatly simplified because a straightforward, one-to-one comparison will detect inconsistencies. This section introduces consistency rules for comparison. The beginning of Section 4 listed the lack of traceability as a major challenge during consistency checking. This section shows how to identify inconsistencies, and in doing so, how to handle missing traceability. In the following, we will first describe the basics of comparison and how to handle traceability under normal conditions. Thereafter, we will discuss ambiguous reasoning to handle missing traceability. In the following we will refer to the *interpretation* as the abstracted (transformed) low-level class diagram and to the *realization* as the existing high-level class diagram. The goal of consistency checking is to compare the realization with the interpretation.

Before transformation, we knew about (some) traceability between the high-level diagram (realization) and the low-level diagram but no traceability is known between the realization and the interpretation. This problem can be fixed easily. Any transformation technique should be able to maintain traceability between the transformation result (interpretation) and the input data (low-level diagram). This is easy because transformation knows what low-level model elements contribute to the interpretation. Through transitive reasoning, we then derive traceability between the realization and the interpretation. For example, we know that the derived *Hotel* is the result of grouping {*HotelEntity*, *Room*, *HotelCollection*} (see dashed arrows in Figure 3) and we know that this group traces to the high-level *Hotel* (mapping table in Figure 2). Thus, there is a transitive trace dependency between the class *Hotel* in the realization and the *Hotel* in the interpretation. Arrows with circular arrowheads in Figure 3 show these transitive trace dependencies between the realization and interpretation.

Ideally, there should be one-to-one traces between realization and interpretation elements only. Unfortunately, partial knowledge about trace dependencies may result in one-to-many dependencies or even many-to-many dependencies (e.g., realization relation *reservation_for* traces to two relationships in the interpretation). This is represented with a fork-shaped trace arrow in Figure 3.

In the following we present a small sample of consistency rules relevant in this paper. Consistency rules have two parts; a qualifier to delimit the model elements it applies to and a condition that must be valid for the consistency to be true[3]:

1. Type of low-level relation is different from abstraction:

```
∀ r ∈ relations, interpretation(r)≠null ⇒ type(interpretation(r))=type(r)
```

Rule 1 states that for a relation to be consistent it must have the same type as its corresponding interpretation. Its qualifier (before "⇒") defines that this rule applies to relations only that have a known interpretation. The traceability arrow in Figure 3 defines such known interpretations (or in reverse known realizations). In Figure 3, we have six interpretation traces; three of which are originating from relationships (circular ends attached to lines): the realization relations "stays_at" and "reservation_for" satisfy above condition[4], however, the realization relation "makes" does not. The latter case denotes an inconsistency because "makes" is of type "aggregation" and its interpretation is of type "association." If we now follow the abstraction traces backward (dashed arrows, that were generated during abstraction), it becomes possible to identify the classes *Account* and *Transaction* as well as their relationships to *GuestEntity* and *PaymentTransaction* a having contributed to the inconsistent interpretation.

2. Low-level relation has no corresponding abstraction:

```
∀ r ∈ relations, abstractions(r)->size=0 ∧ realizations(r)=null ⇒
¬[∃ c ∈ classes(r), realizations(c)≠null]
```

Rule 2 states that all (low-level) relations must trace to at least one high-level model element. To validate this case, the qualifier states that it applies (1) to relations that do not have any *abstractions* (dashed arrows) and (2) to relations that do not have *realizations*. Figure 3 has many relations (derived and user-defined ones). Checking for relations that do not have abstractions ensures that only the most high-level, abstracted relations are considered; ignoring low-level relations such as the aggregation from *Transaction* to *Account*. The rule thus defines that consistency is ensured if *none* of the classes attached to the relation have realizations themselves. The generalization from *Cash* to *Payment* in Figure 3 violates this rule. This generalization neither has an abstraction nor a realization trace but its attached class *Payment*

[3] Some qualifier conditions were omitted for brevity (e.g., checking for transformation type) since they are not needed here.

[4] For now treat the one-to-many traces as two separate one-to-one traces. We will discuss later how to deal with it properly.

has a realization trace[5]. This example implies that the high-level diagram does not represent the relationship to *Cash* or that traceability about it is unknown.

3. Destination direction/navigability of relation does not match abstract relation:

```
∀ r ∈ relations, interpretation(r)≠null ∧
type(interpretation(r))=type(r) ⇒ [size(r->destClass ÷
realization(interpretation(r)->destClass))=0]
```

Rule 3 defines that for two relations to be consistent they ought to be pointing in the same directions (same destination classes). This rule applies to relations that have interpretations and to relations that have the same type. It defines that the realization "r" must have the same destination classes as the realizations of the interpretation's destination classes. A destination class here is a class at the end of a relation's arrowhead (e.g., *Hotel* for *reservation_for*). This rule applies to the two relations *reservation_for* and *stays_at* only (the relation *makes* is ruled out since the qualifier requires relations to be of the same type).

Ambiguous Reasoning

Comparison is not sufficient to establish consistency correctly. Rule 3 applied to the realization relations *reservation_for* and *stays_at* results in consistency being true for both cases. This is misleading because the traceability is ambiguous in that the two high-level (realization) relations point to the same two interpretations (labeled (A) and (B) in Figure 3). The problem is caused by the lack of traceability. Our approach addresses this problem by hypothesizing that at least one of the potentially many choices ought to be consistent. Thus, comparison attempts to find one interpretation for *reservation_for* and one for *stays_at* that is consistent. If no interpretation is consistent then there is a clear inconsistency in the model. If exactly one interpretation is consistent then this interpretation must be the missing trace (otherwise there would be an inconsistency). Finally, if more than one interpretation is consistent then the situation remains ambiguous (although potentially less ambiguous since inconsistent interpretations can still be eliminated as choices). Should more than one consistency rule apply to a model element then all of them need to be satisfied. Each constraint may thus exclude any inconsistent interpretation it encounters.

For instance, in case of the relation *reservation_for*, our approach compares it with both interpretations (A) and (B). It finds rule 3 to be inconsistent if the relation *reservation_for* is compared to interpretation (A); and it finds the rule to be consistent if it is compared to (B). Our approach thus eliminates the trace to interoperation (A) as being incorrect (obviously it leads to inconsistency which cannot be correct). What remains is an ideal, one-to-one mapping. Our approach then does the same for the realization *stays_at* with the result this it is also inconsistent with interpretation (A) and consistent with interpretation (B). Again, the trace to the inconsistent interpretation is removed.

[5] It is outside the scope of this paper to discuss the workings of our reduced redundancy model which treats derivatives like *Payment* together with *PaymentTransaction* as "one element."

Ambiguous reasoning must ensure that every realization has exactly one interpretation it does not share with another realization. For example, in the previous two evaluations we found exactly one interpretation for both realizations *reservation_for* and *stays_at*; however, in both cases it is the same interpretation. This violates one-to-one comparison. Recall that transformation ensures that model elements become directly comparable. Every realization must have exactly one interpretation. To resolve this problem we have to identify the conflicting use of the same interpretations: this is analogous to the resource allocation problem which handles the problem on how to uniquely allocate a resource (resource = interpretation). The maximum flow algorithm [5] (Ford-Fulkerson [12]) solves the resource allocation problem efficiently. The algorithm can be applied to undirected graphs (true in our case since traceability links are undirected) where the algorithm guarantees a maximum matching of edges (traces) without the same vertex (model elements) being used twice. In short, the maximum-bi-partite-matching problem can be used to avoid the duplicate use of interpretations. In the previous example, the algorithm is not able to find a solution that satisfies both realizations. It thus detects an inconsistency.

It must be noted at this point that our ambiguity resolution mechanism has an element of randomness in that the outcome may vary if the order, in which model elements are validated, differs. As such, the maximum bi-partite algorithm will use interpretation (B) for either *stays_at* or *reservation_for* and report a resource conflict (~inconsistency) for the other.

In summary, validating the consistency among model elements potentially encounters three situations as depicted in Figure 4. Situation (a) is the most simplistic one where there is a one-to-one mapping between interpretation (I) and realization (R). The example discussed in Rule 1 above showed such a case. Situation b) corresponds to the example we discussed with Rule 2 where we encountered a low-level interpretation (low-level relation) that had no abstraction. The reverse is also possible where there is a high-level realization that has no refinement. While discussing Rule 3 with its ambiguity example, we encountered situation c) where one realization had two or more interpretations (one-to-many mapping). This scenario required the validation of consistency on all interpretations (OR condition). Traces to inconsistent interpretations were removed and the maximum-partite algorithm was used to find a configuration that resolved all remaining ambiguities randomly.

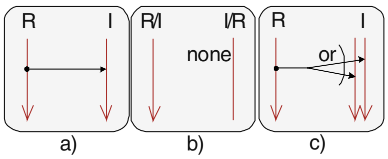


Fig. 4. Basic Comparison Rules for Ambiguity

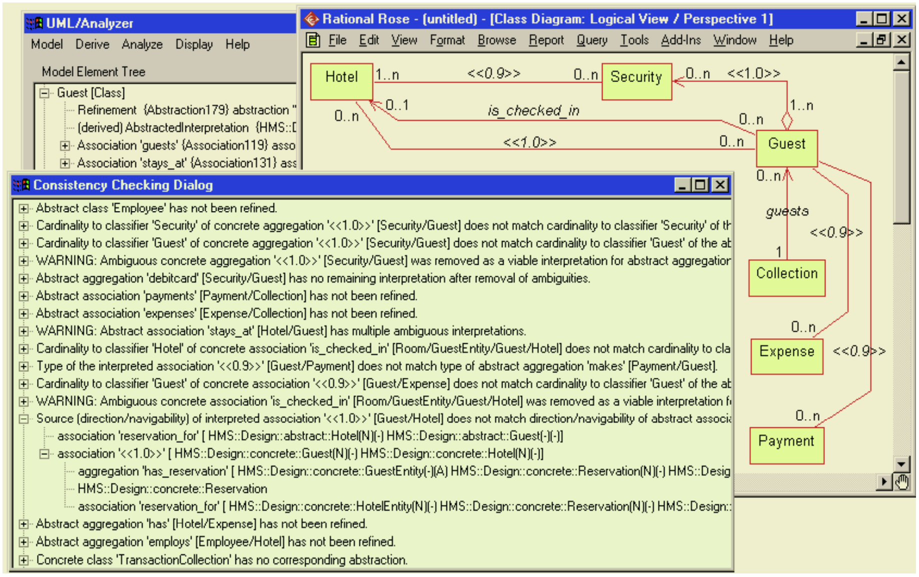
5 Discussion

Scope

In addition to the (in)consistency rules presented in this paper, we identified almost 20 more that apply to refinement [8]. Figure 5 (bottom left) shows an excerpt of inconsistencies between the diagrams in Figure 2 as generated by our tool *UML/Analyzer*. Our tool is also integrated with Rational Rose® which is used as a graphical front-end. The right side depicts the complete derived abstraction of the low-level diagram (Figure 2). Partially hidden in the upper left corner of Figure 5 is the UML/Analyzer main window, depicting the repository view of our example. Besides inconsistency messages, our tool also gives extensive feedback about the model elements involved. For instance, in Figure 5 one inconsistency is displayed in more detail, revealing three low-level model elements (e.g., *Reservation*) as the potential cause of the inconsistency. We also identified around 40 additional inconsistency types between other types of UML diagrams [8] (sequence and state chart diagrams) and the non-UML language C2SADEL [10].

Accuracy (True Inconsistencies/False Inconsistencies)

An important factor on how to estimate the accuracy of any consistency checking approach is in measuring how often it provides erroneous feedback (e.g., report of inconsistencies were there are none or missing inconsistencies). As any automated inconsistency detection approach, our approach may not produce correct results at all times. However, our approach provides means of evaluating the level of "trust" one may have in its feedback. For instance, in Table 1 we presented abstraction rules and


UML/Analyzer
Model Derive Analyze Display Help
Model Element Tree
Rational Rose - (untitled) - [Class Diagram: Logical View / Perspective 1]
File Edit View Format Browse Report Query Tools Add-Ins Window Help
Hotel
Security
Guest
Collection
Expense
Payment
1..n
<<0.9>>
0..n
0..n
<<1.0>>
0..1
is_checked_in
1..n
0..n
<<1.0>>
0..n
0..n
guests
<<0.9>>
1
0..n
<<0.9>>
0..n
Consistency Checking Dialog

Guest [Class]
Refinement {Abstraction179} abstraction "
(derived) AbstractedInterpretation {HMS::D
Association 'guests' {Association119} asso
Association 'stays_at' {Association131} ass

Abstract class 'Employee' has not been refined.
Cardinality to classifier 'Security' of concrete aggregation '<<1.0>>' [Security/Guest] does not match cardinality to classifier 'Security' of th
Cardinality to classifier 'Guest' of concrete aggregation '<<1.0>>' [Security/Guest] does not match cardinality to classifier 'Guest' of the ab
WARNING: Ambiguous concrete aggregation '<<1.0>>' [Security/Guest] was removed as a viable interpretation for abstract aggregatior
Abstract aggregation 'debitcard' [Security/Guest] has no remaining interpretation after removal of ambiguities.
Abstract association 'payments' [Payment/Collection] has not been refined.
Abstract association 'expenses' [Expense/Collection] has not been refined.
WARNING: Abstract association 'stays_at' [Hotel/Guest] has multiple ambiguous interpretations.
Cardinality to classifier 'Hotel' of concrete association 'is_checked_in' [Room/GuestEntity/Guest/Hotel] does not match cardinality to cla
Type of the interpreted association '<<0.9>>' [Guest/Payment] does not match type of abstract aggregation 'makes' [Payment/Guest].
Cardinality to classifier 'Guest' of concrete association '<<0.9>>' [Guest/Expense] does not match cardinality to classifier 'Guest' of the ab
WARNING: Ambiguous concrete association 'is_checked_in' [Room/GuestEntity/Guest/Hotel] was removed as a viable interpretation fc
Source (direction/navigability) of interpreted association '<<1.0>>' [Guest/Hotel] does not match direction/navigability of abstract associa
association 'reservation_for' [HMS::Design::abstract::Hotel(N)(-) HMS::Design::abstract::Guest(-)(-)]
association '<<1.0>>' [HMS::Design::concrete::Guest(N)(-) HMS::Design::concrete::Hotel(N)(-)]
aggregation 'has_reservation' [HMS::Design::concrete::GuestEntity(-)(A) HMS::Design::concrete::Reservation(N)(-) HMS::Desig
HMS::Design::concrete::Reservation
association 'reservation_for' [HMS::Design::concrete::HotelEntity(N)(-) HMS::Design::concrete::Reservation(N)(-) HMS::Design::
Abstract aggregation 'has' [Hotel/Expense] has not been refined.
Abstract aggregation 'employs' [Employee/Hotel] has not been refined.
Concrete class 'TransactionCollection' has no corresponding abstraction.

Fig. 5. UML/Analyzer tool depicting inconsistencies

commented that each rule has a reliability number. Our approach also uses those numbers to derive an overall estimation of how accurate the abstraction is. For example, in Figure 5 we see that our tool derived a high-level *association* between *Security* and *Hotel* and indicated it to be 90% reliable (<<0.9>>) certain that it is correct, indicating high trustworthiness. Another way of indicating accuracy is in the inconsistency feedback itself. For instance, in Figure 5 we see a warning asserting that *stays_at* has multiple ambiguous interpretations followed by another warning indicating that *is_checked_in* was removed as a viable interpretation of *reservation_for*. These warnings indicate that one should also investigate the surrounding elements due to ambiguity. The accuracy of our approach is improved if (1) transformation is more reliable and (2) more trace information is provided.

Scalability
In terms of scalability we distinguish computational complexity and manual intervention. Comparison, our actual consistency checking activity is very fast (O(n)) since it only requires the one-time traversal of all model elements and a simple comparison. Comparison with ambiguous reasoning is also fast since the maximum bi-partite algorithm is computationally linear with respect to the number of model elements. Naturally, transformation is more complex but its scalability can be improved by reusing previously derived model elements (see [9] for a detailed discussion on abstraction scalability). This is something a pure comparative consistency checking approach could never do. To date we have applied our tool on UML models with up to several thousand model elements without problems in computational complexity. More significant, however, is the minimal amount of manual intervention required to use our approach. For a small problem it is quite feasible to provide sufficient human guidance (e.g., more traces, less/no ambiguities), however, for larger systems it is infeasible to expect complete model specifications. In that respect, our approach has the most significant benefits. We already outlined throughout this paper how partial specifications, ambiguities, and even complex many-to-many mappings can be managed successfully by our approach. In case of larger systems this implies substantial savings in human effort and cost since complete specifications are often very hard if not impossible to generate manually [13].

Change Propagation
Model-based software development has the major disadvantage that changes within views have to be propagated to all other views that might have overlapping information. Our consistency checking technique supports change propagation in that it points out places where views differ. The actual process of updating models, however, must still be performed manually. Here, transformation may be used as an automated means of change propagation (see also [9]).

Shortcomings of Transformation
Our approach relies on good transformation techniques. Especially in context of a less-than-perfect modeling language, such as the UML, the reliability of transforma-

tion suffers. Comparison needs to compensate for deficiencies of transformation methods to reduce the number of false positives. For example, our approach conservatively identifies methods of abstracted classes where the true set of methods is a subset of the transformation result. This transformation deficiency can be addressed by comparison checking for a subset of methods instead of the same set. Other deficiencies can be addressed similarly.

6 Related Work

Existing literature uses transformation for consistency checking mostly as a means of converting modeling information into a more precise, formal representation. For instance, VisualSpecs [4] uses transformation to substitute the imprecision of OMT (a language similar to UML) with formal constructs like algebraic specifications followed by analyzing consistency issues in context of that representation; Belkhouche-Lemus [3] follows along the tracks of VisualSpecs in its use of a formal language to substitute statechart and dataflow diagrams; and Van Der Straeten [25] uses description logic to preserve consistency. We also find that formal languages are helpful, however, as this paper demonstrated, we also need transformation methods that “interpret” views in order to reason about ambiguities. Neither of their approaches is capable of doing that. Furthermore, their approaches create the overhead of a third representation.

Grundy et al. took a slightly different approach to transformation in context of consistency checking. In their works on MViews/JViews [14] they investigated consistency between low-level class diagrams and source code by transforming them into a “base model” which is a structured repository. Instead of reasoning about consistency within a formal language, they instead analyze the repository. We adopted their approach but use the standardized UML’s meta model as our repository definition. Furthermore, MViews/JViews does not actually interpret models (like the other approaches above), which severely limits their number of detectable inconsistencies.

Viewpoints [15] is another consistency checking approach that uses inconsistency rules which are defined and validated against a formal model base. Their approach, however, emphasizes more “upsteam” modeling techniques; and has not been shown to work on partial and ambiguous specifications. Nevertheless, Viewpoints also extends our work in that it addresses issues like how to resolve inconsistencies or how to live with them; aspects which are considered outside the scope of this paper.

Koskimies et al. [18] and Keller et al. [17] created transformation methods for sequence and state chart diagrams. It is exactly these kinds of transformations we need; in fact, we adopted Koskimies et al.’s approach as part of ours. Both transformation techniques, however, have the drawback that they were never integrated with a consistency checking approach. This limits their techniques for transformation only. Also, as transformation techniques they have the major drawbacks that extensive specifications and/or human intervention are needed while using them. This is due to the inherent differences between state charts and sequence diagrams. Ehrig et al. [11] also emphasizes model transformation. In their case they take collections of object diagrams and reason about their differences. They also map method calls to changes in their object views, allowing them to reason about the impact methods have. Their approach has, however, only been shown to work for a single type of view and they also have also not integrated their approach into a consistency checking framework.

Our work also relates to the field of transformational programming [20,22]. We have proposed a technique that allows systematic and consistent refinement of models that, ultimately, may lead to code. The main differences between transformational programming and our approach are in the degrees of automation and scale. Transformational programming is fully automated, though its applicability has been demonstrated primarily on small, well-defined problems [22]. Our refinement approach, on the other hand, can be characterized only as semi-automated; however, we have applied it on larger problems and a more heterogeneous set of models, typical of real development situations.

SADL [21] follows a different path in formal transformation and consistency. This approach makes use of a proof-carrying formal language that enables consistent refinement without human intervention. The SADL approach is very precise, however, has only been shown to work on their language. It remains unknown whether a more heterogeneous set of models can be also refined via this approach. Also, the SADL approach has only been used for small samples using small refinement steps.

Besides transformation, another key issue of consistency checking is the traceability across modeling artifacts. Traceability is outside the scope of this work but, as this paper has shown, it is very important. Capturing traces is not trivial, as researchers have recognized [13], however, there are techniques that give guidance. Furthermore, process modeling is also outside the scope, although we find it very important in the context of model checking and transformation. To date, we have shown that a high degree of automation is possible, but have not reached full automation yet. Processes are important since they must take over wherever automation ends [19,23].

7 Conclusion

This paper presented a transformation-based consistency checking approach for consistent refinement and abstraction. Our approach separates model validation into the major *Mapping (Traceability)*, *Transformation*, and *Comparison* which may be applied iteratively throughout the software development life cycle to adapt and evolve software systems. To date, our approach has been applied successfully to a number of third party models including the validation of a part of a Satellite Telemetry Processing, Tracking, and Commanding System (TT&C) [2], the Inter-Library Loan System [1] as well as several reverse-engineered tools (including UML/Analyzer itself).

We invented and validated our abstraction technique in collaboration with Rational Software. Our consistency checking approach is fully automated and tool supported. Our approach is also very lightweight since it does not require the use of third-party (formal) languages [4,15,21] but instead integrates seamlessly into existing modeling languages. We demonstrated this in context of the Unified Modeling Language and C2SADEL.

References

1. Abi-Antoun, M., Ho, J., and Kwan, J. Inter-Library Loan Management System: Revised Life-Cycle Architecture. 1999.
2. Alvarado, S.: "An Evaluation of Object Oriented Architecture Models for Satellite Ground Systems," *Proceedings of the 2nd Ground Systems Architecture Workshop (GSAW)*, February 1998.
3. Belkhouche, B. and Lemus, C.: "Multiple View Analysis and Design," *Proceedings of the Viewpoint 96: International Workshop on Multiple Perspectives in Software Development*, October 1996.
4. Cheng, B. H. C., Wang, E. Y., and Bourdeau, R. H.: "A Graphical Environment for Formally Developing Object-Oriented Software," *Proceedings of IEEE International Conference on Tools with AI*, November 1994.
5. Cormen, T.H., Leiserson, C. E., Rivest, R. L.: Introduction to Algorithms. MIT Press, 1996.
6. Egyed A.: A Scenario-Driven Approach to Trace Dependency Analysis. *IEEE Transactions on Software Engineering (TSE)* 29(2), 2003, 116-132.
7. Egyed, A.: "Compositional and Relational Reasoning during Class Abstraction," *Proceedings of the 6th International Conference on the Unified Modeling Language (UML)*, San Francisco, USA, October 2003.
8. Egyed, Alexander. Heterogeneous View Integration and its Automation. PhD Dissertation, Technical Report from the University of Southern California, USA, 2000.
9. Egyed A.: Automated Abstraction of Class Diagrams. *ACM Transaction on Software Engineering and Methodology (TOSEM)* 11(4), 2002, 449-491.
10. Egyed, A. and Medvidovic, N.: "A Formal Approach to Heterogeneous Software Modeling," *Proceedings of 3rd Foundational Aspects of Software Engineering (FASE)*, March 2000, pp.178-192.
11. Ehrig H., Heckel R., Taentzer G., and Engels G.: A Combined Reference Model- and View-Based Approach to System Specification. *International Journal of Software Engineering and Knowledge Engineering* 7(4), 1997, 457-477.
12. Ford, L.R., Fulkerson, D. R.: Flows in Networks. Princeton University Press, 1962.
13. Gieszl, L. R.: "Traceability for Integration," *Proceedings of the 2nd Conference on Systems Integration (ICSI 92)*, 1992, pp.220-228.
14. Grundy J., Hosking J., and Mugridge R.: Inconsistency Management for Multiple-View Software Development Environments. *IEEE Transactions on Software Engineering (TSE)* 24(11), 1998.
15. Hunter, A. and Nuseibeh, B.: "Analysing Inconsistent Specifications," *Proceedings of 3rd International Symposium on Requirements Engineering (RE97)* , January 1997.
16. Jackson, J.: "A Keyphrase Based Traceability Scheme," *IEE Colloquium on Tools and Techniques for Maintaining Traceability During Design*, 1991, pp.2-1-2/4.
17. Khriss, I., Elkoutbi, M., and Keller, R.: "Automating the Synthesis of UML Statechart Diagrams from Multiple Collaboration Diagrams," *Proceedings for the Conference of the Unified Modeling Language*, June 1998, pp.132-147.
18. Koskimies K., Systä T., Tuomi J., and Männistö T.: Automated Support for Modelling OO Software. *IEEE Software*, 1998, 87-94.
19. Lerner, B. S., Sutton, S. M., and Osterweil, L. J.: "Enhancing Design Methods to Support Real Design Processes," *IWSSD-9*, April 1998.

20. Liu, J., Traynor, O., and Krieg-Bruckner, B.: "Knowledge-Based Tranformational Programming," *4th International Conference on Software Engineering and Knowledge Engineering*, 1992.
21. Moriconi M., Qian X., and Riemenschneider R. A.: Correct Architecture Refinement. *IEEE Transactions on Software Engineering* 21(4), 1995, 356-372.
22. Partsch H. and Steinbruggen R.: Program Transformation Systems. *ACM Computing Surveys* 15(3), 1983, 199-236.
23. Perry, D. E.: "Issues in Process Architecture," *9th International Software Process Workshop*, Airlie, VA, October 1994.
24. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison Wesley, 1999.
25. Van Der Straeten, R., Mens, T., Simmonds, J. and Jonckers, V.: “Using Description Logic to Maintain Consistency between UML Models,” Proceedings of 6th International Conference on the Unified Modeling Language, San Francisco, USA, 2003, pp. 326-340..

Measuring Aspect Cohesion

Jianjun Zhao[1] and Baowen Xu[2]

[1] Department of Computer Science and Engineering, Fukuoka Institute of Technology
3-30-1 Wajiro-Higashi, Fukuoka 811-0295, Japan
zhao@cs.fit.ac.jp

[2] Department of Computer Science and Engineering, Southeast University
Nanjing 210096, China
bwxu@cs.seu.edu.cn

Abstract. Cohesion is an internal software attribute representing the degree to which the components are bound together within a software module. Cohesion is considered to be a desirable goal in software development, leading to better values for external attributes such as maintainability, reusability, and reliability. Aspect-oriented software development (AOSD) is a new technique to support separation of concerns in software development. AOSD introduces a new kind of component called *aspect* which is like a class, also consisting of attributes (aspect instance variables) and those modules such as advice, introduction, pointcuts, and methods. The cohesion for such an aspect is therefore mainly about how tightly the attributes and modules of aspects cohere. To test this hypothesis, cohesion measures for aspects are needed. In this paper, we propose an approach to assessing the aspect cohesion based on dependence analysis. To this end, we present various types of dependencies between attributes and/or modules in an aspect, and the *aspect dependence graph* (ADG) to explicitly represent these dependencies. Based on the ADG, we formally define some aspect cohesion measures. We also discuss the properties of these dependencies, and according to these properties, we prove that these measures satisfy the properties that a good measure should have.

1 Introduction

Aspect-oriented software development (AOSD) is a new technique to support separation of concerns in software development [2,12,13,14]. The techniques of AOSD make it possible to modularize crosscutting aspects of a system. Aspects in AOSD may arise at any stage of the software life cycle, including requirements specification, design, implementation, etc. Some examples of crosscutting aspects are exception handling, synchronization, and resource sharing.

The current research so far in AOSD is focused on problem analysis, software design, and implementation techniques. However, efficient evaluations of this new design technique in a rigorous and quantitative fashion are still ignored during the current stage of the technical development. For example, it has been frequently claimed that applying an AOSD method will eventually lead to quality

M. Wermelinger and T. Margaria-Steffen (Eds.): FASE 2004, LNCS 2984, pp. 54–68, 2004.
© Springer-Verlag Berlin Heidelberg 2004

software, but unfortunately, there is little data to support such claim. Aspect-oriented software is supposed to be easy to maintain, reuse, and evolve, yet few quantitative studies have been conducted, and measures to quantify the amount of maintenance, reuse, and evolution in aspect-oriented systems are lacking. In order to verify claims concerning the maintainability, reusability, and reliability of systems developed using aspect-oriented techniques, software measures are required.

As with procedural and object-oriented software, we should also relate aspect-oriented structural quality to some critical process attributes concerning software maintainability, reusability, and reliability. We therefore need appropriate measures of aspect-oriented structure to begin to relate structure to process. Recently, Zhao developed a suite of dependence-based structural measures which are specifically designed to quantify the information flows in aspect-oriented software [16].

Cohesion is a structural attribute whose importance is well-recognized in the software engineering community [4,8,15]. Cohesion is an internal software attribute representing the degree to which the components are bound together within a software module. Cohesion is considered to be a desirable goal in software development, which may lead to better values for external attributes such as maintainability, reusability, and reliability. In procedural or object-oriented paradigm, a highly cohesive component is one with one basic function and is difficult to be decomposed. Cohesion is therefore considered to be a desirable goal in software development, leading to better values for external attributes such as maintainability, reusability, and reliability. A system should have high cohesion. Recently, many cohesion measures and several guidelines to measure cohesion of a component have been developed for procedural software [3,11] and for object-oriented software [6,8,10,4,5].

Aspect-oriented language introduces a new kind of component called *aspect* to model the crosscutting concerns in a software system. An aspect with its encapsulation of state (attributes) and associated modules (operations) such as advice, introduction, pointcuts, and methods is a different abstraction in comparison to a procedure within procedural systems and a class within object-oriented systems. The cohesion of an aspect is therefore mainly about how tightly the aspect's attributes and modules cohere.

However, although cohesion has been studied widely for procedural and object-oriented software, it has not been studied for aspect-oriented software yet. Since an aspect contains new modules such as advice, introduction, and pointcuts that are different from methods in a class, existing class cohesion measures can not be directly applied to aspects. Therefore, new measures that are appropriate for measuring aspect cohesion are needed.

In this paper, we propose an approach to assessing the aspect cohesion based on dependence analysis. To this end, we present various types of dependencies between attributes and/or modules such as advice, introduction, pointcuts, and methods of an aspect, and a dependence-based representation called *aspect dependence graph* (ADG) to represent these dependencies. Based on the ADG, we formally define some aspect cohesion measures. We also discuss the properties of these dependencies, and according to these properties, we prove that these measures satisfy the properties that a good measure should have.

We hope that by studying the ideas of aspect cohesion from several different viewpoints and through well developed cohesion measures, we can obtain a better understanding of what the cohesion is meant in aspect-oriented paradigm and the role that cohesion plays in the development of quality aspect-oriented software. As the first step to study the aspect cohesion, the goal of this paper is to provide a sound and formal basis for aspect cohesion measures before applying them to real aspect-oriented software design.

The rest of the paper is organized as follows. Section 2 briefly introduces AspectJ, a general aspect-oriented programming language based on Java. Section 3 defines three types of dependencies in an aspect and discusses some basic properties of these dependencies. Section 4 proposes an approach to measuring aspect cohesion from three facets: inter-attribute, module-attribute and inter-module. Section 5 discusses some related work. Concluding remarks are given in Section 6.

2 Aspect-Oriented Programming and AspectJ

We present our basic ideas of aspect cohesion measurement approach for aspect-oriented programs in the context of AspectJ, the most widely used aspect-oriented programming language [1]. Our basic ideas, however, deal with the basic concepts of aspect-oriented programming and therefore apply to the general class of aspect-oriented languages.

AspectJ [1] is a seamless aspect-oriented extension to Java. AspectJ adds to Java some new concepts and associated constructs. These concepts and associated constructs are called join point, pointcut, advice, introduction, and aspect.

Aspect is a modular unit of crosscutting implementation in AspectJ. Each aspect encapsulates functionality that crosscuts other classes in a program. An aspect is defined by aspect declaration, which has a similar form of class declaration in Java. Similar to a class, an aspect can be instantiated and can contain state and methods, and also may be specialized in its sub-aspects. An aspect is then combined with the classes it crosscuts according to specifications given within the aspect. Moreover, an aspect can *introduce* methods, attributes, and interface implementation declarations into types by using the *introduction* construct. Introduced members may be made visible to all classes and aspects (public introduction) or only within the aspect (private introduction), allowing one to avoid name conflicts with pre-existing members.

The essential mechanism provided for composing an aspect with other classes is called a *join point*. A join point is a well-defined point in the execution of a program, such as a call to a method, an access to an attribute, an object initialization, exception handler, etc. Sets of join points may be represented by *pointcuts*, implying that such sets may crosscut the system. Pointcuts can be composed and new pointcut designators can be defined according to these combinations. AspectJ provides various pointcut *designators* that may be combined through logical operators to build up complete descriptions of pointcuts of interest. For a complete listing of possible designators one can refer to [1].

```
public class Point {
  protected int x, y;
  public Point(int _x, int _y) {
    x = _x;
    y = _y;
  }
  public int getX() {
    return x;
  }
  public int getY() {
    return y;
  }
  public void setX(int _x) {
    x = _x;
  }
  public void setY(int _y) {
    y = _y;
  }
  public void printPosition() {
    System.out.println("Point
                at("+x+","+y+")");
  }
  public static void main(String[] args) {
    Point p = new Point(1,1);
    p.setX(2);
    p.setY(2);
  }
}
class Shadow {
  public static final int offset = 10;
  public int x, y;

  Shadow(int x, int y) {
    this.x = x;
    this.y = y;
  public void printPosition() {
    System.outprintln("Shadow at
          ("+x+","+y+")");
  }
}
```

```
aspect PS_Protocol {
  private int shadowCount = 0;
  public static int getCount() {
    return PS_Protocol.aspectOf().shadowCount;
  }
  private Shadow Point.shadow;
  public static void associate(Point p, Shadow s){
    p.shadow = s;
  }
  public static Shadow getShadow(Point p) {
    return p.shadow;
  }

  pointcut setting(int x, int y, Point p):
    args(x,y) && call(Point.new(int,int));
  pointcut settingX(Point p):
    target(p) && call(void Point.setX(int));
  pointcut settingY(Point p):
    target(p) && call(void Point.setY(int));

  after(int x, int y, Point p) returning :
    setting(x, y, p) {
    Shadow s = new Shadow(x,y);
    associate(p,s);
    shadowCount++;
  }
  after(Point p): settingX(p) {
    Shadow s = new getShadow(p);
    s.x = p.getX() + Shadow.offset;
    p.printPosition();
    s.printPosition();
  }
  after(Point p): settingY(p) {
    Shadow s = new getShadow(p);
    s.y = p.getY() + Shadow.offset;
    p.printPosition();
    s.printPosition();
  }
}
```

Fig. 1. A sample AspectJ program.

An aspect can specify *advice* that is used to define some code that should be executed when a pointcut is reached. Advice is a method-like mechanism which consists of code that is executed *before*, *after*, or *around* a pointcut. Around advice executes *in place* of the indicated pointcut, allowing a method to be replaced.

An AspectJ program can be divided into two parts: *base code* part which includes classes, interfaces, and other language constructs for implementing the basic functionality of the program, and *aspect code* part which includes aspects for modeling crosscutting concerns in the program. Moreover, any implementation of AspectJ should ensure that the base and aspect code run together in a properly coordinated fashion. Such a process is called *aspect weaving* and involves making sure that applicable advice runs at the appropriate join points. For detailed information about AspectJ, one can refer to [1].

Example. Fig. 1 shows an AspectJ program that associates shadow points with every `Point` object. The program contains one aspect `PS_Protocol` and two classes `Point` and `Shadow`. The aspect has three methods `getCount`, `associate` and `getShadow` and three pieces of advice related to pointcuts `setting`, `settingX` and `settingY` respectively[1]. The aspect also has two attributes, i.e., `shadowCount` which is an attribute of the aspect itself and `shadow` which is an attribute that is privately introduced to class `Point`.

[1] Since advice in AspectJ has no name. So for easy expression, we use the name of a pointcut to stand for the name of advice it associated with.

In the rest of the paper, we use this example to demonstrate our basic ideas of aspect cohesion measurement. We also assume that an aspect is composed of attributes (aspect instance variables), and modules[2] such as advice, introduction, pointcuts and methods.

3 Aspect Dependencies

In this section we define various types of dependencies between modules and/or attributes in an aspect and discuss some properties of these dependencies.

3.1 Dependence Definitions

We define three types of dependencies between attributes and/or modules in an aspect, that is, *inter-attribute*, *inter-module*, and *module-attribute dependence.*

Definition 1. *Let a_1, a_2 be attributes in an aspect. a_2 is* inter-attribute *dependent on a_1, denoted by $a_2 \hookrightarrow a_1$, if one of the following conditions holds:*

- *The definition of a_2 uses (refers) a_1 directly or indirectly, or*
- *Whether a_2 can be defined is determined by the state of a_1.*

Generally, if a_2 is used in the condition part of a control statement (such as `if` and `while`), and the definition of a_1 is in the inner statement of the control statement, then the definition of a_1 depends on the state of a_2. For example, according to Definition 1, we know that there is no inter-attribute dependence in aspect `PS_Protocol` of Fig. 1.

There are two types of dependencies between aspect modules: *inter-module call dependence* and *inter-module potential dependence.*

Definition 2. *Let m_1, m_2 be two modules and a be an attribute in an aspect. m_2 is* inter-module-dependent *on m_1, denoted by $m_2 \rightarrow m_1$, if one of the following conditions holds:*

- *m_1 is called in m_2. (inter-module call dependence)*
- *a is used in m_2 before it is defined, and a is defined in m_1. (inter-module potential dependence)*

Given an aspect, we can not assume which piece of introduction or which method in the aspect might be invoked before another[3]. So we assume that all the introduction and methods in the aspect can be invoked at any time and in any order. Therefore, if m_2 might use an attribute a, and a is defined in m_1, and if m_1 is invoked first and then m_2 is invoked, then m_2 might use a defined in m_1, i.e. m_2 is inter-module potentially-dependent on m_1.

[2] For unification, we use the word "module" to stand for a piece of advice, a piece of introduction, a pointcut, or a method declared in an aspect.

[3] In AspectJ, advice is automatically woven into some methods in a class by the compiler, and therefore no call exists for the advice.

To obtain the inter-module dependencies, for each module m, we introduce two sets, D_{IN} and D_{OUT}, where $D_{IN}(m)$ is the set of attributes referred before modifying their values in m, and $D_{OUT}(m)$ is the set of attributes modified in m. Thus, for an attribute a and modules m and m', if $a \in D_{IN}(m') \cap D_{OUT}(m)$, then $m' \rightarrow m$.

In addition to attributes, since there are four types of different modules in an aspect, i.e., advice, introduction, pointcuts and methods, there may have the following possible types of inter-module dependencies, i.e., dependencies between advice and advice, advice and introduction, advice and method, advice and pointcut[4], introduction and introduction, introduction and method, or method and method.

Example. In order to compute inter-module dependencies in aspect `PS_Protocol`, we first compute the D_{IN} and D_{OUT} sets for each module in `PS_Protocol`. They are: $D_{IN}(\texttt{getCount}) = \{\texttt{shadowCount}\}$, $D_{OUT}(\texttt{getCount}) = \emptyset$, $D_{IN}(\texttt{getShadow}) = \{\texttt{shadow}\}$, $D_{OUT}(\texttt{getShadow}) = \emptyset$, $D_{IN}(\texttt{associate}) = \emptyset$, $D_{OUT}(\texttt{associate}) = \{\texttt{shadow}\}$, $D_{IN}(\texttt{setting}) = \{\texttt{shadowCount}\}$, $D_{OUT}(\texttt{setting}) = \{\texttt{shadowCount}\}$, $D_{IN}(\texttt{settingX}) = D_{OUT}(\texttt{settingX}) = \emptyset$, $D_{IN}(\texttt{settingY}) = D_{OUT}(\texttt{settingY}) = \emptyset$. Also there exist inter-module dependencies between each pointcut and its corresponding advice. So we finally get the following inter-module dependencies in `PS_Protocol`.
(method `getCount` $\rightarrow$ advice`setting`), (advice `setting` $\rightarrow$ method `associate`), (advice `settingX` $\rightarrow$ method `getShadow`), (advice `settingY` $\rightarrow$ advice `getShadow`), (pointcut `setting` $\rightarrow$ advice `setting`), (pointcut `settingX` $\rightarrow$ advice `settingX`), and (pointcut `settingY` $\rightarrow$ advice `settingX`).

Definition 3. *Let m be a module and a be an attribute in an aspect. m is* module-attribute-dependent *on a, denoted by $m \mapsto a$, if a is referred in m.*

Since there are four types of different modules in an aspect, module-attribute dependencies may have four different types: *advice-attribute*, *introduction-attribute*, *pointcut-attribute*, or *method-attribute* dependencies.

Example. According to Definition 3, the module-attribute dependencies in aspect `PS_Protocol` are: (method `getCount` $\mapsto$ attribute `shadowCount`), (method `getShadow` $\mapsto$ attribute `shadowCount`), (advice `settingX` $\mapsto$ attribute `shadowCount`).

Note that all these types of dependencies defined above can be derived by performing control flow and data flow analysis using existing flow analysis algorithms [17]. Due to the space limitation, we do not discuss this issue here.

3.2 Dependence Properties

This section discusses some properties of dependencies defined in Section 3.1, and refines the definition of inter-module dependence to fit for measuring aspect cohesion.

[4] A pointcut is only related to its corresponding advice. Therefore, there is no dependence between pointcut and method, pointcut and introduction, or pointcut and pointcut.

According to definition 1, if $a_1 \hookrightarrow a_2$ and $a_2 \hookrightarrow a_3$, then $a_1 \hookrightarrow a_3$. Therefore, we have

Property 1. The inter-attribute dependencies are transitive.

Based on Property 1, we can define the inter-attribute transitive dependence as follows.

Definition 4. *Let A be an aspect and a_i $(i > 0)$ be attributes in A. If there exist attributes $a_1, a_2, \ldots, a_{n-1}, a_n$ $(n > 1)$, where $a_1 \hookrightarrow a_2$, …, $a_{i-1} \hookrightarrow a_i$, …, $a_{n-1} \hookrightarrow a_n$, then a_1 is* inter-attribute-transitive-dependent *on a_n, denoted by $a_1 \overset{*}{\hookrightarrow} a_n$.*

According to definition 2, for modules m_1, m_2, and m_3, if $m_1 \rightarrow m_2$ and $m_2 \rightarrow m_3$, then $m_1 \rightarrow m_3$ may not hold. Consider an example of inter-module call dependence, if m_1 is called in m_2 and m_2 is called in m_3, then m_1 is not necessarily called in m_3. For inter-module potential dependence, if $m_1 \rightarrow m_2$ and $m_2 \rightarrow m_3$ are introduced by unrelated, different attributes, then m_1 might have no relation with m_3. Therefore, we have

Property 2. The inter-module dependencies are not transitive.

The intransitivity among inter-module dependencies leads to great difficulties when performing analysis. Thus, we should redefine the inter-module dependencies.

Definition 5. *Let m_1, m_2 be modules and a be an attribute in an aspect. If a is used in m_1 and defined in m_2, then a used in m_1 is dependent on a defined in m_2, denoted by $m_1 \xrightarrow{a,a} m_2$, where <a, a> is named as a tag.*

For unification, add a tag <*, *> for each inter-module call dependence arc, i.e., if m_1 is inter-module-call-dependent on m_2, then we have $m_1 \xrightarrow{*,*} m_2$.

Definition 5 is the basic definition. Since the dependencies between attributes are transitive, we can obtain a more general definition according to Property 3.

To obtain such dependencies, we introduce two sets for each module m of an aspect, i.e., D_A and D_{AO}, each element of which has the form (a, a'), where a and a' are attributes of the aspect.

– $D_A(m)$ is the set of dependencies which records the dependencies from the attributes referred in m to the attributes defined out m. $D_A(m)$ is a subset of inter-attribute dependencies.
– $D_{AO}(m)$ is the set of dependencies which records the dependencies from the attributes referred in m to the attribute defined out m when exiting m.

In general, the intermediate results are invisible outside, and an attribute might be modified many times in a piece of advice, a piece of introduction, or a method. We introduce D_{AO} to improve the precision. Obviously, we have $D_{AO}(m) \subseteq D_A(m)$.

Definition 6. *Let m_1, m_2 be modules and a_1, a_2 be attributes in an aspect. If $(a_1, a_2) \in D_A(m_1)$ and $a_2 \in D_{OUT}(m_2)$, then m_1 is dependent on m_2, denoted by $m_1 \xrightarrow{a_1,a_2} m_2$.*

According to Definition 6, we have the following properties:

Property 3. Let m_1, m_2, m_3 be modules and a_1, a_2, a_3 be attributes in an aspect. If $m_1 \xrightarrow{a_1,a_2} m_2$, and $\forall(a_2, a_3)$, $(a_2, a_3) \in D_{AO}(m_2)$ and $a_3 \in D_{OUT}(m_3)$, then $m_1 \xrightarrow{a_1,a_3} m_3$.

Since $D_{AO}(m_1) \subseteq D_A(m_1)$, according to Definition 6, if $(a_2, a_3) \in D_{AO}(m_1)$, and $a_3 \in D_{OUT}(m_2)$, then $m_1 \xrightarrow{a_1,a_2} m_3$. Thus, we have Corollary 1.

Corollary 1. *Let m_1, m_2, m_3 be modules and a_1, a_2, a_3 be attributes in an aspect. If $m_1 \xrightarrow{a_1,a_2} m_2$ and $m_2 \xrightarrow{a_2,a_3} m_3$, then $m_1 \xrightarrow{a_1,a_3} m_3$.*

Property 4. Let m_1, m_2, m_3 be modules and a_1, a_2 be attributes in an aspect. If $m_1 \xrightarrow{*,*} m_2$ and $m_2 \xrightarrow{a_1,a_2} m_3$, then $m_1 \xrightarrow{a_1,a_2} m_3$.

From Properties 2-4, we can define the inter-module transitive dependence as follows.

Definition 7. *Let A be an aspect, m_i $(i > 0)$ be modules, and a_i $(i > 0)$ be attributes in A. If there exist modules $m_1, \ldots, m_n$ and attributes $a_1, \ldots, a_n$ $(n > 1$, a_i need not be unique, and a_i may be "$*$", which models calls between modules), where $m_1 \xrightarrow{a_1,a_2} m_2, \ldots, m_{i-1} \xrightarrow{a_{i-1},a_i} m_i, \ldots, m_{n-1} \xrightarrow{a_{n-1},a_n} m_n$, then m_1 is* inter-module-transitive-dependent *on m_n, denoted by $m_1 \xrightarrow{*} m_n$.*

To present our cohesion measure in a unified model, we introduce the aspect dependence graph to explicitly represent all types of dependencies in an aspect.

Definition 8. *The aspect dependence graph (ADG) of an aspect A is a directed graph[5], $G_{ADG} = (V, A, T)$ where $V = V_a \cup V_m$, $A = A_{aa} \cup A_{mm} \cup A_{ma}$, and $T \in A \times (V', V')$ are the sets of vertex, arc, and tag respectively, such that:*
- *V_a is the set of attribute vertices: each represents a unique attribute (the name of a vertex is the name of the attribute it represents) in A.*
- *V_m is the set of module vertices: each represents a unique module (the name of a vertex is the name of the module it represents) in A.*
- *V' is the union of V_a and $\{*\}$, i.e., $V' = V_a \cup \{*\}$.*
- *A_{aa} is the set of inter-attribute dependence arcs that represents dependencies between attributes, such that for v_a, $v_{a'} \in V_a$ if $a \hookrightarrow a'$, then $(v_a, v_{a'}) \in A_{aa}$.*
- *A_{mm} is the set of inter-module dependence arcs that represent dependencies between modules, such that for v_m, $v_{m'} \in V_m$ if $m \to m'$, then $(v_m, v_{m'}) \in A_{mm}$.*

[5] A directed graph $G = (V, A)$, where V is a set of vertices and $A \in V \times V$ is a set of arcs. Each arc $(v, v') \in A$ is directed from v to v'; we say that v is the source and v' the target of the arc.

– *A_{ma} is the set of module-attribute dependence arcs that represents dependencies between modules and attributes, such that for $v_m \in V_m$, $v_a \in V_a$ if $m \mapsto a$, then $(v_m, v_a) \in A_{ma}$.*

Generally, G_{ADG} consists of three sub-graphs, i.e., *inter-attribute dependence graph* $G_{AAG} = (V_a, A_{aa})$, *module-attribute dependence graph* $G_{MAG} = (V, A_{ma})$, and *inter-module dependence graph* $G_{MMG} = (V_m, A_{mm}, T)$, which can be used to define the *inter-attribute*, *module-attribute*, and *inter-module* cohesion in an aspect respectively. Fig. 2 shows the G_{MAG} and G_{MMG} of aspect `PS_Protocol`. Since there exists no inter-attribute dependence in the aspect, the G_{AAG} is not available. Note that we omit the Tags in both graphs for convenience.

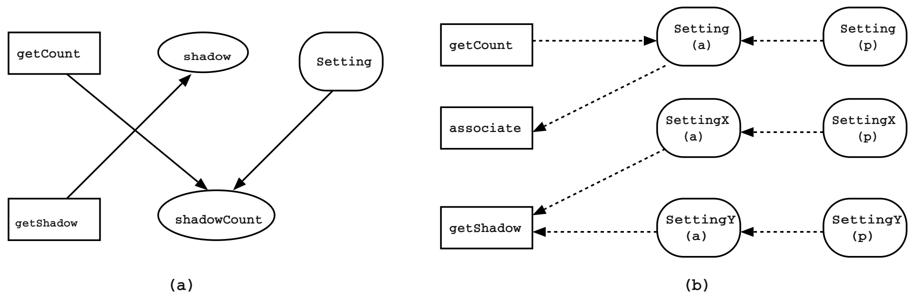


Fig. 2. The G_{MAG} (a) and G_{MMG} (b) of the aspect `PS_Protocol` in Fig. 1.

4 Measuring Aspect Cohesion

Briand *et al.* [4] have stated that a good cohesion measure should have properties such as *non-negative and standardization*, *minimum and maximum*, *monotony*, and *cohesion does not increase when combining two components*. We believe that these properties provide also a useful guideline even in aiding the development of an aspect cohesion measure. In this section, we propose our aspect cohesion measures, and show that our aspect cohesion measures satisfy the properties given by Briand *et al.* [4].

An aspect consists of attributes and modules such as advice, introduction, pointcuts, and methods. There are three types of dependencies between attributes and/or modules. Thus, the cohesion of an aspect should be measured from the three facets. In the following discussion, we assume that an aspect A consists of k attributes and n modules, where k, $n \geq 0$.

4.1 Measuring Inter-attribute Cohesion

Inter-attribute cohesion is about the tightness between attributes in an aspect. To measure the inter-attribute cohesion for an aspect A, for each attribute a of A, we introduce a set D_a to record the attributes on which a depends, i.e., $D_a(a) = \{a \mid a_1 \overset{*}{\hookrightarrow} a, a_1 \neq a\}$. Thus, we define the inter-attribute cohesion of A as:

$$\gamma_a(A) = \begin{cases} 0 & \text{k} = 0 \\ 1 & \text{k} = 1 \\ \frac{1}{k}\sum_{i=1}^{k} \frac{|D_a(a_i)|}{k-1} & k > 1 \end{cases}$$

where k is the number of attributes in A, and $\frac{|D_a(a_i)|}{k-1}$ represents the degree on which a_i depends on other attributes in A.

If $k = 0$, there is no attribute in A. Inter-attribute cohesion is useless, thus we set $\gamma_a(A) = 0$. If $k = 1$, there is only one attribute in A. Although it can not depend on other attribute, it itself is tight, thus we set $\gamma_a(A) = 1$. If each attribute relates to all others, then $\gamma_a(A) = 1$. If all attributes can exist independently, then $\gamma_a(A) = 0$. Thus, $\gamma_a(A) \in [0, 1]$.

Theorem 1. *Let A be an aspect and $G_{AAG} = (V_a, A_{aa})$ be the inter-attribute dependence graph of A. $\gamma_a(A)$ does not decrease when adding an arc $(a_1, a_2) \in A_{aa}$, where $a_1, a_2 \in V_a$, on G_{AAG}.*

Proof: Let $D_a(a_1)$ be the set of attributes in A which a_1 depends on before adding an arc. We use $D'_a(a_1)$ to represent $D_a(a_1)$ after adding an arc (a_1, a_2) to $D_a(a_1)$.

(1) If $a_2 \in D_a(a_1)$, according to the definition of D_a, a_1 transitively depends on a_2, $D_a(a_1)$ does not change when adding arc (a_1, a_2), i.e., $D_a(a_1) = D'_a(a_1)$, $\gamma_a(A)$ keeps unchanged;
(2) If $a_2 \notin D_a(a_1)$, according to the definition of D_a, $D'_a(a_1)$ will increase after adding arc (a_1, a_2), i.e., $D'_a(a_1) = D_a(a_1) \cup \{a_2\}$. For other attributes that depend on a_1, they will transitively depend on a_2, after adding arc (a_1, a_2). In all, $\gamma_a(A)$ will increase.

Therefore, $\gamma_a(A)$ does not decrease when adding an arc on G_{AAG}.

Theorem 2. *Let A_1 and A_2 be two aspects and A_{12} be an aspect derived from the combination of A_1 and A_2. Let $\gamma_a(A_1)$ and $\gamma_a(A_2)$ be the inter-attribute cohesions of A_1 and A_2 respectively and $\gamma_a(A_{12})$ be the inter-attribute cohesion of A_{12}. $\gamma_a(A_{12}) \le max\{\gamma_a(A_1), \gamma_a(A_2)\}$.*

Proof: When combining the two aspects, the previous dependencies do not change in the new aspect, and there is no new dependence added. Let A_1 and A_2 be two aspects that have k_1 and k_2 attributes respectively.

(1) If $k_1 = k_2 = 0$, then $\gamma_a(A_{12}) = \gamma_a(A_1) = \gamma_a(A_2) = 0$.
(2) If $k_1 = 1$ or $k_2 = 1$, we assume $k_1 = 1$, then $\gamma_a(A_1) = 1$. Because this is the maximum of the cohesion, $\gamma_a(A_{12})$ is no greater than $\gamma_a(A_1)$.
(3) If k_1, $k_2 > 1$, $k_1 + k_2 > k_1$ and $k_1 + k_2 > k_2$. For each attribute a_1 of A_1, we have $\frac{|D_a(a_1)|}{k_1-1} \ge \frac{|D_a(a_1)|}{k_1+k_2-1}$. For each attribute a_2 of A_2, we have $\frac{|D_a(a_2)|}{k_2-1} \ge \frac{|D_a(a_2)|}{k_1+k_2-1}$. Thus, $\gamma_a(A_{12}) \le max\{\gamma_a(A_1), \gamma_a(A_2)\}$.

In all the cases above, inter-attribute cohesion does not increase when combining two aspects.

Example. The D_a sets for each module in `PS_Protocol` are: $D_a(\texttt{shadowCount}) = D_a(\texttt{shadow}) = \emptyset$. Therefore, $\gamma_a(\texttt{PS\_Protocol}) = \frac{1}{2}\sum_{i=1}^{2} \frac{|D_a(a_i)|}{2-1} = 0$.

4.2 Measuring Module-Attribute Cohesion

Module-attribute cohesion is about the tightness between modules and attributes in an aspect. To measure this kind of cohesion, for each module m in an aspect A, we introduce two sets: D_{ma} and D^o_{ma}, where

- $D_{ma}(m)$ is the set of all A's attributes that are referred in m.
- $D^o_{ma}(m)$ is a set of all A's attributes that are referred in m and related to attributes referred in other modules, i.e., $D^o_{ma}(m) = \{a \mid \exists a_1, m_1 \text{ such that } ((m_1 \xrightarrow{a_1,a} m) \vee (m \xrightarrow{a,a_1} m_1)) \wedge (a, a_1 \neq {}'*')\}$.

Obviously, $D^o_{ma}(m) \subseteq D_{ma}(m)$. We can define the module-attribute cohesion for A as follows:

$$\gamma_{ma}(A) = \begin{cases} 0 & \text{n} = 0 \\ 1 & \text{n} = 1 \text{ and } |D_{ma}(m_i)| \neq 0 \\ \frac{1}{n}\sum_{i=1}^{n} \frac{|D^o_{ma}(m_i)|}{|D_{ma}(m_i)|} & \text{others} \end{cases}$$

where n is the number of modules in A, and $\frac{|D^o_{ma}(m_i)|}{|D_{ma}(m_i)|}$, denoted by $\rho(m_i)$, is the ratio between the number of attributes which are referred in m_i and relevant to others, to the number of all attributes referred in m_i.

For a module m, if $D_{ma}(m) = \emptyset$, i.e., no attribute is referred in m, we set $\rho(m) = 0$. If the attributes referred in m are not related to other modules, these attributes can work as local variables. It decreases the cohesion to take a local variable for a module as an attribute for all modules. If there is no attribute or module in the aspect, no module will depend on others. There is no D_{ma} or all the D_{ma} are empty, i.e., $|D_{ma}(m)| = 0$. Thus, $\gamma_{ma} = 0$. If each attribute referred in m is related to other modules, then $\rho(m) = 1$.

Theorem 3. *Let A be an aspect and $G_{MMG} = (V_m, A_{mm})$ be the inter-module dependence graph of A. Let m_1 be a module of A and $\rho(m_1) = \frac{|D^o_{ma}(m_1)|}{|D_{ma}(m_1)|}$. $\rho(m_1)$ does not decrease when adding an arc (m_1, m_2), where $m_1, m_2 \in V_m$, on G_{MMG}.*

Proof: Let the arc added have the form $m_1 \xrightarrow{a_1,a_2} m_2$, the attribute set m_1 refers is $D_{ma}(m_1)$ before adding the arc, among which the attributes related to other modules are included in the set $D^o_{ma}(m_1)$. They change to $D'_{ma}(m_1)$ and $D^o_{ma}{}'(m_1)$ after adding the arc.

(1) If $a \in D^o_{ma}(m_1)$, according to the definitions, since the two sets do not change when adding an arc, $\rho(m_1)$ does not change.

(2) If $a \notin D^o_{ma}(m_1)$ and $a \in D_{ma}(m_1)$, $D_{ma}(m_1)$ will not be changed when adding the arc, but $D^o_{ma}(m_1)$ will be increased, i.e. $D^o_{ma}{}'(m_1) = D^o_{ma}(m_1) \cup \{a\}$. Thus $\rho(m_1)$ will increase.

(3) If $a \notin D_{ma}(m_1)$, since $D^o_{ma}(m_1) \subseteq D_{ma}(m_1)$, $a \notin D^o_{ma}(m_1)$. Therefore, the two sets will increase after adding the arc, i.e., $D^o_{ma}{}'(m_1) = D^o_{ma}(m_1) \cup \{a\}$, and $D'_{ma}(m_1) = D_{ma}(m_1) \cup \{a\}$. Then $|D^o_{ma}{}'(m_1)| = |D^o_{ma}(m_1)| + 1$, and $|D'_{ma}(m_1)| = |D_{ma}(m_1)| + 1$. Since $\frac{|D^o_{ma}(m_1)|}{|D_{ma}(m_1)|} \leq \frac{|D^o_{ma}(m_1)|+1}{|D_{ma}(m_1)|+1}$, $\rho(m_1)$ does not decrease.

(4) If the added arc is an inter-module call dependence arc, we have $D^o_{ma}{}'(m_1) = D^o_{ma}(m_1) \cup D^o_{ma}{}'(m_2)$ and $D'_{ma}(m_1) = D_{ma}(m_1) \cup D_{ma}(m_2)$. Applying these relations to cases (1) - (3), we will have the same conclusions.

Similarly, we can prove the conclusion holds if the arc added is $m_2 \overset{a_1,a_2}{\longrightarrow} m_1$.

Example. The D_{ma} and D^o_{ma} sets for each module in PS_Protocol are D_{ma}(getShadow) = {shadow}, D_{ma}(getCount) = D_{ma}(setting) = {shadowCount}, D_{ma}(associate) = D_{ma}(settingX) = D_{ma}(settingY) = $\emptyset$, D^o_{ma}(getCount) = D^o_{ma}(getShadow) = $\emptyset$, D^o_{ma}(associate) = D^o_{ma}(setting) = $\emptyset$, and D^o_{ma}(settingX) = D^o_{ma}(settingY) = $\emptyset$. Therefore, γ_{ma}(PS_Protocol) $= \frac{1}{6}\sum_{i=1}^{6} \frac{|D^o_{ma}(m_i)|}{|D_{ma}(m_i)|} = 0$.

4.3 Measuring Inter-module Cohesion

In the G_{MMG}, although the modules can be connected by attributes, this is not necessary sure that these modules are related. If there does exist some relations between modules, we should determine their tightness. This is the process to measure the inter-module cohesion. To do this, we introduce another set D_m for each module m in an aspect A, where $D_m(m) = \{m_2 \mid m_1 \overset{*}{\longrightarrow} m_2\}$.

The inter-module cohesion $\gamma_m(A)$ for A is defined as follows:

$$\gamma_m(A) = \begin{cases} 0 & \text{n} = 0 \\ 1 & \text{n} = 1 \\ \frac{1}{n}\sum_{i=1}^{n} \frac{|D_m(m_i)|}{n-1} & n > 1 \end{cases}$$

where n is the number of modules in A and $\frac{|D_m(m_i)|}{n-1}$ represents the tightness between m_i and other modules in A. If each module depends on all other modules, then $\gamma_m(A) = 1$. If all modules are independent, i.e., each module has no relation with any other modules, then $\gamma_m(A) = 0$.

Theorem 4. *Let A be an aspect, $G_{MMG} = (V_m, A_{mm})$ be the inter-module dependence graph of A. The inter-module cohesion $\gamma_m(A)$ does not decrease when adding an arc $(m_1, m_2) \in A_{mm}$, where $m_1, m_2 \in V_m$, on G_{MMG}.*

Theorem 5. *Let A_1 and A_2 be aspects and A_{12} be an aspect derived from the combination of A_1 and A_2. Let $\gamma_m(A_1)$ and $\gamma_m(A_2)$ be the inter-module cohesions of A_1 and A_2 and $\gamma_m(A_{12})$ be the inter-module cohesion of A_{12}. $\gamma_m(A_{12}) \le max\{\gamma_m(A_1), \gamma_m(A_2)\}$.*

We can prove Theorems 4 and 5 with a similar way as we did for Theorems 1 and 2. Due to the limitation of the space, we do not repeat them here.

Example. The D_m sets for each module in PS_Protocol are: D_m(getCount) = {setting}, D_m(getShadow) = {shadow}, D_m(setting) = {associate}, D_m(settingX) = {getShadow}, and D_m(settingY) = {getShadow}. Therefore, γ_m(PS_Protocol) = $\frac{1}{6}\sum_{i=1}^{6} \frac{|D_m(m_i)|}{|6-1|} = \frac{1}{6}$.

4.4 Measuring Aspect Cohesion

After measuring the three facets of aspect cohesion independently, we have a discrete view of the cohesion of an aspect. We have two ways to measure the aspect cohesion for an aspect A:

(1) Each measurement works as a field. The aspect cohesion for A is a 3-tuple as $\Gamma(A) = (\gamma_a, \gamma_{ma}, \gamma_m)$.
(2) Integrating the three facets as a whole. Let $x = \beta_1 * \gamma_a + \beta_2 * \gamma_{ma} + \beta_3 * \gamma_m$, the aspect cohesion for A is computed as follows.

$$\Gamma(A) = \begin{cases} 0 & \text{n} = 0 \\ \beta * \gamma_m & \text{k} = 0 \text{ and } n \neq 0 \\ x & \text{others} \end{cases}$$

where k is the number of attributes and n is the number of modules in A, $\beta \in (0, 1]$, $\beta_1, \beta_2, \beta_3 > 0$, and $\beta_1 + \beta_2 + \beta_3 = 1$.

If $k = 0$ and $n \neq 0$, $\Gamma(A)$ describes only the tightness of the call relations, thus we introduce a parameter β to constrain it. For other cases, we introduce three parameters β_1, β_2, and β_3 to constrain it. The selection of β_1, β_2, and β_3 is determined by users.

Example. The aspect cohesion of `PS_Protocol` can be computed based on its γ_a, γ_{ma}, and γ_m. If we set $\beta_1 = \beta_2 = \beta_3 = \frac{1}{3}$, we have $\Gamma(\texttt{PS\_Protocol}) = \frac{1}{3} * \gamma_a(\texttt{PS\_Protocol}) + \frac{1}{3} * \gamma_{ma}(\texttt{PS\_Protocol}) + \frac{1}{3} * \gamma_m(\texttt{PS\_Protocol}) = \frac{1}{18}$.

5 Related Work

We discuss some related work that directly or indirectly influences our work presented in this paper. To the best of our knowledge, our work is the first attempt to study how to assess the cohesion of aspects in aspect-oriented software.

The approaches taken to measure cohesiveness of procedural programs have generally tried to evaluate cohesion on a procedure (function) by procedure (function) basis. Bieman and Ott [3] propose an approach to measuring the cohesion on procedures based on a relation between output tokens (output variables) and program slices. Kang and Bieman [11] investigate to measure cohesion at the design level for the case that the code has yet to be implemented. Since aspects are more complex and significantly different abstractions in comparing with procedures (functions), These measures definitely fails to be applied to aspects.

Most existing approaches for class cohesion measurement consider the interactions between methods and/or attributes in a class. Chidamber and Kemerer [8] propose the *Lack of Cohesion Measure* (LCOM) to assess class cohesion based on the similarity of two methods in a class. Hitz and Montazeri [10] propose an extension to the LCOM of Chidamber and Kemerer by making it more sensitive to small changes in the structure of a class. Chae, Kwon, and Bae [6] propose a class cohesion measure for object-oriented system by introducing a new notion called *glue methods*. In contrast to the above cohesion measurement approaches that only consider the interaction between methods and/or attributes, Chen *et al.* [7] propose an approach to measuring class cohesion based on the interactions between attributes and/or methods in a class. Although their work is similar to ours, we see our work differing from theirs because our approach can handle interactions between attributes and those modules such as aspect advice, introduction, and pointcuts that are unique constructs for aspect-oriented programs.

Based on these new types of interactions we propose a new dependence model for aspects that is different from the dependence model presented by Chen *et al.* [7].

Zhao [16] proposes a metrics suite for aspect-oriented software, which are specifically designed to quantify the information flows in aspect-oriented programs. To this end, Zhao presents a dependence model for aspect-oriented software which is composed of several dependence graphs to explicitly represent dependence relationships in a module, a class, or the whole program. Based on the model, Zhao defines some metrics to measure the complexity of an aspect-oriented program from various different viewpoints and levels. However, Zhao does not address the issue of aspect cohesion measurement.

The development of coupling measures for aspect-oriented software is also considered by Zhao who proposes an approach to assessing the coupling of aspect-oriented software based on the interactions between aspects and classes [18].

Dufour *et al.* [9] proposes some dynamic metrics for AspectJ, which focuses on the performance and execution time costs, rather than structural complexity of aspect-oriented software.

6 Concluding Remarks

In this paper, we proposed an approach to measuring the cohesion of aspects in aspect-oriented software based on dependence analysis. We discussed the tightness of an aspect from three facets: *inter-attribute*, *module-attribute* and *inter-module*. These three facets can be used to measure the aspect cohesion independently and also can be integrated as a whole. We also discussed the properties of these dependencies and according to these properties we proved that our cohesion measures satisfy some properties which a good measure should have. Therefore, we believe our approach may provide a solid foundation for measuring aspect cohesion.

The aspect cohesion measures proposed in this paper focused only on the features of an aspect itself, and did not take its application environment into account. When do so, there will be a little difference because the modules in the aspect may be invoked in a set of given sequences, which is a subset of the arbitrary combination. For such a case, we should analyze the definitions and uses of attributes in the context of the applications. Also, in this paper we did not distinguish the connected and non-connected graphs and did not consider to measure the cohesion of a derived aspect (i.e., aspect inheritance). In our future work, we will study the influence of aspect inheritance and other aspect-oriented features on aspect cohesion, and apply our cohesion measure approach to real aspect-oriented software design.

Acknowledgments

This work was partially supported by the Japan Society for Promotion of Science (JSPS) under Grand-in-Aid for Scientific Research (C) (No.15500027). The authors would like to thank the anonymous referees for valuable comments on an earlier version of the paper.

References

1. The AspectJ Team. The AspectJ Programming Guide. 2002.
2. L. Bergmans and M. Aksits. Composing crosscutting Concerns Using Composition Filters. *Communications of the ACM*, Vol.44, No.10, pp.51-57, October 2001.
3. J. Bieman and L. Ott. Measuring Functional Cohesion. *IEEE Transactions on Software Engineering*, Vol.22, No.10, pp.644-657, August 1994.
4. L.C. Briand, J. Daly and J. Wuest. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering*, Vol.3, No.1, pp.65-117, 1998.
5. L.C. Briand, S. Morasca and V.R. Basili. Defining and Validating Measures for Object-Based High-Level Design. *IEEE Transactions on Software Engineering*, Vol.25, No.5, pp.724-743, 1999.
6. H.S. Chae, Y. R. Kwon and D. H. Bae. A Cohesion Measure for Object-Oriented Classes. *Software Practice & Experience*, Vol.30, No.12, pp.1405-1431, 2000.
7. Z. Chen, Y. Zhou, B. Xu, J. Zhao, and H. Yang. A Novel Approach to Measuring Class Cohesion Based on Dependence Analysis. *Proceedings of the International Conference on Software Maintenance*, October 2002.
8. S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, Vol.20, No.6, pp.476-493, 1994.
9. B. Dufour, C. Goard, L. Hendren, C. Verbrugge, O. de Moor, and G. Sittampalam. Measuring the Dynamic Behavior of AspectJ Programs. Scable Technical Report No.2003-8, McGill University and Oxford University, December 2003.
10. M. Hitz and B. Montazeri. Measuring Coupling and Cohesion in Object-Oriented Systems. *Proceedings of International Symposium on Applied Corporate Computing*, pp.25-27, Monterrey, Mexico, October 1995.
11. B. Kang and J. Bieman. Design-Level Cohesion Measures: Derivation, Comparison, and Applications. Computer Science Technical Report CS-96-103, Colarado State University, 1996.
12. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. Aspect-Oriented Programming. *Proceedings of the 11th European Conference on Object-Oriented Programming*, pp.220-242, LNCS, Vol.1241, Springer-Verlag, June 1997.
13. K. Lieberher, D. Orleans, and J. Ovlinger. Aspect-Oriented Programming with Adaptive Methods. *Communications of the ACM*, Vol.44, No.10, pp.39-41, October 2001.
14. P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. *Proceedings of the International Conference on Software Engineering*, pp.107-119, 1999.
15. E. Yourdon and L. Constantine. Structural Desing. Englewood Cliffs, Prentice Hall, 1979.
16. J. Zhao. Toward A Metrics Suite for Aspect-Oriented Software. Technical Report SE-136-5, Information Processing Society of Japan (IPSJ), March 2002.
17. J. Zhao and M. Rinard. System Dependence Graph Construction for Aspect-Oriented Programs. Technical Report MIT-LCS-TR-891, Laboratory for Computer Science, MIT, March 2003.
18. J. Zhao. Coupling Measurement in Aspect-Oriented Systems. Technical Report SE-142-6, Information Processing Society of Japan (IPSJ), June 2003.

Refactoring Object-Z Specifications

Tim McComb

School of Information Technology and Electrical Engineering
The University of Queensland
tjm@itee.uq.edu.au

Abstract. Object-Z offers an object-oriented means for structuring formal specifications. We investigate the application of refactoring rules to add and remove structure from such specifications to forge object-oriented designs. This allows us to tractably move from an abstract functional description of a system toward a lower-level design suitable for implementation on an object-oriented platform.

1 Introduction

Previous research added object orientation to specification languages to take advantage of the modularity and structure benefits that this approach offers. Whilst this has been achieved with formalisms such as Object-Z [22], VDM^{++} [11], and Z^{++} [11], which are built upon existing well-defined languages, there is not yet a practical process for turning these functional specifications into well-structured object-oriented software. This seems like an intuitively appropriate goal for such specification languages, as their semantics already closely model that of object-oriented programming languages.

There are two major aspects such a process must encompass. First, at a high level, a purely functional specification must be reorganised and refined to create a reasonable object-oriented design. Second, at a lower level, the object-oriented design must be further refined to object-oriented code.

We focus on the higher level objective: moving from an abstract functional specification in Object-Z to a specification of an object-oriented design. We achieve this by systematically modifying the specification to introduce design, taking advantage of rules that are similar to the *refactoring* [6,18] rules presented by Fowler [6]. Although we borrowed the term, our approach slightly differs from the normal interpretation of refactoring as the process of improving an existing design.

Fowler extensively catalogued object-oriented refactoring rules, but made no attempt to formalise the rules or the process of introducing them. Cornelio et al. [4], however, specified a set of Fowler's refactoring steps as rules in the ROOL programming language which were verified using a weakest precondition semantics.

Our idea differs from these approaches as we move from an abstract specification to an object-oriented design without the need to consider irrelevant detail at the programming-language level. Also, the ROOL language does not have a reference-based model of object identity. Object references are necessary to implement certain design patterns and architectures and are an integral part of many widely-used object-oriented programming languages like Java and C^{++}. In addition to this, our approach differs from others as it has the primary purpose of introducing a design into a specification rather than simply modifying an existing implementation.

In this paper we show that significant structural object-oriented design can be introduced at the specification level, in Object-Z, utilising two rules. One rule adds structure to the specification, whilst another rule removes structure. When combined with non-structural refactoring steps our approach a powerful method for the incremental introduction of specification-based structural design in Object-Z. Since the rules are behaviour-preserving, this process ensures that any software design produced will be correct with respect to the original specification.

2 Related Work

Refactoring is a relatively new idea as applied to object-oriented programming languages, but the systematic transformation of formal specifications to resemble programs, particularly in the non-object-oriented domain, is certainly not a new idea [13,17].

However, formal and incremental class reorganisation in object-oriented languages was discussed by Casais [3] and also by Mikhajlova and Sekerinski [15], who propose a class refinement semantics for object-oriented specification languages based upon data refinement. This involves the interpretation of subclasses as subtypes, this semantic definition going beyond the Object-Z interpretation of inheritance as strongly syntax-based [21]. This approach, however, does simplify refinement a great deal — as in VDM^{++} and Z^{++} [11] — but restricts the ease of expression in the languages because it inhibits reuse of syntactic definition.

Other investigations into modifying class structure in a formal context were made by Bonsangue et al. [1,2]. Here, object orientation was modelled by extending the action system formalism to support objects and classes. This formalism differs from Mikhajlova and Sekerinski's, even though it has the same interpretation of inheritance, as it considers the addition of methods when subclassing. However, it still considers inheritance as a form of subtyping.

There is significant work in this field in the VDM^{++} language, where Lu [12] initially proposed that classes could be decomposed as a refinement step to introduce structure. Goldsack and Lano [7,10] built upon this work to introduce and formalise *annealing* for decomposing classes in a VDM^{++} specification. The problem of invariant distribution in the decomposition process became the focus of their work, and a complete approach to specification-based object-oriented design was not developed.

3 Motivating Example: Battleship

The game of "Battleship" was chosen as a motivating example. Two players are required, and the game begins by presenting each player with a blank grid consisting of twelve rows and twelve columns. Each player then secretly places a fleet of five ships on the grid. Each ship is a different length ranging from occupying one cell to five cells, and may be placed either horizontally or vertically.

Players then alternate attempts to sink their opponent's ships by posing a query to their opponent as to whether a certain cell in the opponent's playing grid is occupied by a ship or not (this is analogous to firing a missile). Their opponent responds with either: *hit*, *miss*, *ship sunk*, or *fleet destroyed*. A ship is considered sunk when every cell it occupies in the grid has been guessed by the opposing player. A fleet is considered destroyed when all five ships are sunk and the game ends.

In Figure 1 we present a specification of the game written in Object-Z. The class *System* represents the two players in terms of their guesses ($guessesP1$ and

$$
\begin{array}{l}
COORDINATE == 1\,..\,12 \times 1\,..\,12 \\
SHIP ::= destroyer \mid submarine \mid cruiser \mid battleship \mid carrier \\
RESPONSE ::= hit \mid miss \mid sink \mid fleet\_destroyed \\
FLEETBOARD == COORDINATE \pinj SHIP \\
GUESSBOARD == COORDINATE \pinj RESPONSE
\end{array}
$$

System

$$\upharpoonright(\mathrm{INIT}, InitFleetP1, InitFleetP2, FireAtP1, FireAtP2)$$

$$
\begin{array}{l}
guessesP1 : GUESSBOARD \\
guessesP2 : GUESSBOARD \\
fleetP1 : FLEETBOARD \\
fleetP2 : FLEETBOARD
\end{array}
$$

INIT

$$
\begin{array}{l}
guessesP1 = \varnothing \\
guessesP2 = \varnothing \\
fleetP1 = \varnothing \\
fleetP2 = \varnothing
\end{array}
$$

InitFleetP1

$$
\begin{array}{l}
\Delta(fleetP1) \\
fleet? : FLEETBOARD \\
\hline
fleetP1' = fleet?
\end{array}
$$

InitFleetP2

$$
\begin{array}{l}
\Delta(fleetP2) \\
fleet? : FLEETBOARD \\
\hline
fleetP2' = fleet?
\end{array}
$$

FireAtP1

$$
\begin{array}{l}
\Delta(fleetP1, guessesP2) \\
guess? : COORDINATE \\
\hline
guess? \notin \operatorname{dom} guessesP2 \\
fleetP1' = fleetP1 \ndres \{guess?\} \\
guessesP2' = \\
\quad guessesP2 \cup \{guess? \mapsto \\
\quad\quad \text{if } fleetP1' \subset fleetP1 \text{ then} \\
\quad\quad\quad \text{if } \operatorname{ran} fleetP1' \subset \operatorname{ran} fleetP1 \text{ then} \\
\quad\quad\quad\quad \text{if } fleetP1' = \varnothing \text{ then} \\
\quad\quad\quad\quad\quad fleet\_destroyed \\
\quad\quad\quad\quad \text{else } sink \text{ else } hit \text{ else } miss\}
\end{array}
$$

FireAtP2

$$
\begin{array}{l}
\Delta(fleetP2, guessesP1) \\
guess? : COORDINATE \\
\hline
guess? \notin \operatorname{dom} guessesP1 \\
fleetP2' = fleetP2 \ndres \{guess?\} \\
guessesP1' = \\
\quad guessesP1 \cup \{guess? \mapsto \\
\quad\quad \text{if } fleetP2' \subset fleetP2 \text{ then} \\
\quad\quad\quad \text{if } \operatorname{ran} fleetP2' \subset \operatorname{ran} fleetP2 \text{ then} \\
\quad\quad\quad\quad \text{if } fleetP2' = \varnothing \text{ then} \\
\quad\quad\quad\quad\quad fleet\_destroyed \\
\quad\quad\quad\quad \text{else } sink \text{ else } hit \text{ else } miss\}
\end{array}
$$

Fig. 1. Initial monolithic Battleship specification.

$guessesP2$) and fleet positions ($fleetP1$ and $fleetP2$). Each player provides the initial placement of their ships on the game board using the $InitFleeetP1$ and $InitFleetP2$ operations. The $FireAtP1$ and $FireAtP2$ operations specify the behaviour of each player taking a turn. The delta-lists, i.e., lists of the form $\Delta(\ldots)$, identify which state variables the operations may change; all other variables are unchanged.

Our specification is adequate to describe the core functionality of the game, but does not represent an appropriate object-oriented design. A good object-oriented design, and indeed a better structured specification, would identify players as separate objects in the system and encapsulate their functionality; thus reducing the clear redundancy. This paper describes a means for achieving this goal via the step-wise application of behaviour-preserving rules.

We shall introduce, in Section 4, the general rules for modifying the structure of an Object-Z specification. Using these rules, we illustrate their effectiveness in Sections 5 and 6, showing their application to reflect different design decisions whilst preserving the specified behaviour.

4 Structural Modification Rules

Our rules below describe syntactic transformations on Object-Z specifications. The rules rely on class simulation [5] to establish refinement relations between the two sides of the rules to ensure the preservation of behaviour. Refinement is interpreted with respect to a class's external interface, and we allow for the interface of a class to be widened in the refinement process.

We have proven both of the following rules correct by showing that the right-hand side is a refinement of the left-hand side using the definitional rules of Object-Z provided by Smith [22] together with Derrick and Boiten's class simulation laws [5]. Proof sketches are provided in Section 4.3.

For a complete approach, it is necessary to consider other rules that do not modify class structure. Such rules rename variables or operations in classes, simplify or reorganise operation expressions, or involve other equivalence transformations that are proven sound using the language definition. These rules are not elaborated upon for purposes of brevity, but are based upon those presented by Cornelio et al. [4] and Fowler [6].

In the following sections we present the Annealing rule for introducing class structure, and the Coalescence rule for eliminating class structure.

4.1 Annealing

The annealing rule allows for the decomposition of one class into two, effectively partitioning its data and functionality. It is similar in intention to Fowler's *Extract Class* refactoring [6,4], which performs the same function at the programming language level. The annealing rule is fundamental for introducing structure as part of the process of moving from specification towards design. In Figure 2, class A on the left-hand side is annealed to create the classes A_1 and B on the right-hand side.

A

$\upharpoonright(\textsc{Init}, OpA, OpB)$

S
T

$\textsc{Init}$
$R_{\{S,T\}}$

$OpA \mathrel{\widehat=} [\Delta(S) \mid P_{\{S,S',T\}}]$
$OpB \mathrel{\widehat=} [\Delta(T) \mid Q_{\{S,T,T'\}}]$

A_1

$\upharpoonright(\textsc{Init}, S, OpA, OpB)$

S
$component : B$
Δ
T

$\theta T = \theta component.T$
$component.frame = self$

$\textsc{Init}$
$R_{\{S,T\}}$
$component.\textsc{Init}$

$OpA \mathrel{\widehat=} [\Delta(S) \mid P_{\{S,S',T\}}]$
$OpB \mathrel{\widehat=} component.OpB$

B

$\upharpoonright(\textsc{Init}, frame, OpB)$

T
$frame : A_1$
Δ
S

$\theta S = \theta frame.S$

$\textsc{Init}$
$R_{\{S,T\}}$

$OpB \mathrel{\widehat=} [\Delta(T) \mid Q_{\{S,T,T'\}}]$

Fig. 2. Annealing rule.

One class, called the *framing* class (represented in Figure 2 by A_1), holds a reference to an object of the *component* class (represented in Figure 2 by B). The interface of the original class (class A in Figure 2) is extended by the framing class, and it is the responsibility of the framing class to invoke operations upon, and manage the state of, the component class. Any references to the original class from the rest of the system are replaced by references to the new framing class.

The intention is to divide the state $(S;\ T)$ of the class A between two classes: one with state S and one with state T. S and T represent schemas that conjoin to form the state of class A. The predicates P, Q, and R have subscripts indicating the sets of variables that may appear free in them.

Ancestors of the class A are inherited by the newly formed classes A_1 and B. The actual set of classes inherited in each case corresponds with the partition (S or T) that the inherited classes contribute to. Any inherited classes that share a common ancestor must contribute to the same partition. This way, after the rule application, A_1 and B inherit only the classes that contribute to their respective states. Any class in the specification that was a descendant of the original class A must now inherit both A_1 and B.

A further precondition for performing the annealing rule is that every operation in the class to be partitioned *explicitly* changes, if any variables at all, either variables declared in S or variables declared in T, but not both (illustrated by the $\Delta(S)$ and $\Delta(T)$ notation). Operations that make no state changes may appear in either or both classes wherever referenced.

To achieve this precondition concerning operations, any operation schema that changes variables in both S and T must be split (using schema operators) such that the predicates that change variables in S are in a different operation from those that change a variable in T. This can be achieved in most specifications by promoting logical operators to schema operators, and moving schemas into new, separate operations and referencing the new operations from the original one. This is in the spirit of the *Extract Method* [6] refactoring step which splits a programming language procedure into two, where one contains a procedure call to the other.

For example, in Object-Z:

$$X \mathrel{\widehat{=}} \left[\, \Delta(S, T) \mid P_{\{S,S',T\}} \land Q_{\{S,T,T'\}} \,\right]$$

is equivalent to, by promoting logical conjunction to schema conjunction:

$$X \mathrel{\widehat{=}} \left[\, \Delta(S) \mid P_{\{S,S',T\}} \,\right] \land \left[\, \Delta(T) \mid Q_{\{S,T,T'\}} \,\right]$$

which, by introducing an operation reference, is equivalent to:

$$\begin{gathered} X \mathrel{\widehat{=}} \left[\, \Delta(S) \mid P_{\{S,S',T\}} \,\right] \land Y \\ Y \mathrel{\widehat{=}} \left[\, \Delta(T) \mid Q_{\{S,T,T'\}} \,\right] \end{gathered}$$

Operations that modify variables in S must remain in class A_1, and operations that modify variables in T must be moved to the component class B. Operations that are moved into the component class must, in a similar way to the *Move Method* [6] refactoring step, be redefined in the framing class to dereference the component operation, e.g.

$$Y \mathrel{\widehat{=}} \mathit{component}.Y$$

Operation references need to be contained only in the operations that change S (as above). This is so the references can be modified to dereference *component* upon application of the rule.

In the case where predicates cannot be readily split to cater for the segmentation of state, input and output variables may be added to the operation schemas to communicate variable values across the schema boundary using Object-Z's parallel ($\|$) operator. The parallel composition operator binds together output variables on one side (suffixed by !) with input variables on the other side (suffixed by ?) where the variables have the same base name.

The default action of the rule is to make all variables that were accessible in the original class visible to both classes in the substituted system. In Figure 2, this is represented by the inclusion of the respective foreign schema in the states of A_1 and B underneath the delta (Δ) symbol. Any declaration that appears underneath the delta in the state of an Object-Z class may have its value changed by any operation. The state invariant, for example $\theta T = \theta \mathit{component}.T$ in class A_1, ensures that the declared schema T is always synchronised with variable bindings of the coupled object's state $\mathit{component}.T$. This does create a high degree of coupling as references are required in both directions, however unreferenced variables may be removed after application of the rule eliminating unnecessary coupling between the two classes.

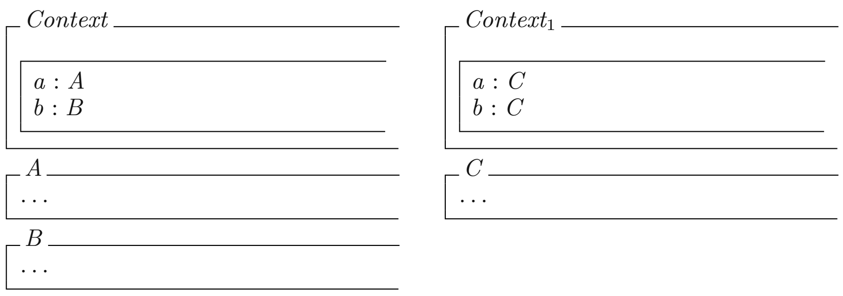


Fig. 3. Coalescence rule, assuming $A \sqsubseteq C$ and $B \sqsubseteq C$.

4.2 Coalescence

Coalescence (see Figure 3) is a rule that allows two classes A and B to be replaced with one class C, so long as A and B are both refined by C (denoted $A \sqsubseteq C$ and $B \sqsubseteq C$ respectively).

This approach allows the designer to extract common functionality from possibly independent parts of the system, perhaps using annealing, and then coalesce this functionality to form a common library or object-type. Because annealing is unidirectional and structure adding, this rule is important for structure reduction.

4.3 Proof Sketches of Rules

Annealing. To prove that the annealing rule is behaviour preserving (i.e., a refinement), we adopt Derrick and Boiten's [5] approach to class simulation.

For a class simulation to exist, that is, for the classes A_1 and B to together simulate the behaviour of class A, it is required that each operation in the system containing A_1 and B is *applicable* whenever the corresponding operation from class A is applicable. Also, we must show that the application of any operation in A_1 and B is *correct* with respect to the corresponding operation in class A, where correctness is defined as adherence to a *retrieve* relation upon states.

For the annealing rule Figure 2, the retrieve relation, which relates the states of A_1 (which indirectly includes B) with A, is defined as such:

$$R \mathrel{\widehat{=}} \left[\, A.\textsc{State};\ A_1.\textsc{State} \mid\ \theta A.S = \theta A_1.S \wedge \theta A.T = \theta A_1.T \,\right]$$

To establish a class simulation, the applicability and correctness of every operation in A_1 needs to be shown with relation to the corresponding operation in A. The class simulation laws provide the necessary proof obligations. We demonstrate the use of these proof obligations for OpA in the appendix.

The arguments for the applicability and correctness of OpB are similar, but for purposes of brevity are not presented.

Coalescence. The coalescence rule is proven by showing that C in Figure 3 is a refinement of both A and B, and is therefore substitutable for both. When the rule is applied these refinements must be shown to hold, except for the trivial case where A is exactly the same as B. This trivial case always holds as a class is always a refinement of itself. Therefore, if two classes are identical, each is a refinement of the other and may thus be replaced by a single class which is identical to both.

5 Restructuring for Passive Players

In this section we systematically add a *Player* class to the monolithic *System* class in Figure 1 using the rules presented in Section 4.

Annealing

We first apply annealing steps to extract the functionality of the two players into two separate classes. This requires two applications of the rule, so the state variables of *System* must be partitioned into two groups, each group representing the data for a single player. These groups combined contain the entire state information for *System*; consisting of $guessesP1$ with $fleetP1$, and $guessesP2$ with $fleetP2$.

The initialisation condition is quite easily distributed between the two classes as it is a simple conjunction of predicates relating to independent parts of the state. The annealing rule requires that every operation in *System* that changes variables in both partitions be split, adding operation references where necessary. $InitFleetP1$ and $InitFleetP2$ require no change as this requirement already holds, but both *FireAt* operations need to be altered.

Progress can be made using logical operator promotions on $FireAtP1$ (and similarly $FireAtP2$), but the conjunct in $FireAtP1$ responsible for updating the $guessesP2$ variable cannot be split by simply promoting conjunction since it also contains references to $fleetP1$ and $fleetP1'$. The parallel composition operator ($\|$) in Object-Z can be used to achieve this end.

The schema containing the aforementioned conjunct should pertain exclusively to player two's partition, as this is what it changes. The required information contained in $fleetP1$ and $fleetP1'$ can be exported from the other schema that modifies player one's partition by introducing two new output variables $fleetP1!$ and $fleetP1'!$ to the schema and equating them to these variables. When input variables are added to the schema that references player two's variable set, (i.e., $fleetP1?$ and $fleetP1'?$), the required variable values may be shared between the two schemas via a parallel composition operator. It is relevant to note that this process of introducing a parallel operator over schema conjunction is completely syntactic and automatable. The resultant operation is:

$$
\begin{aligned}
&\mathit{FireAtP1} \mathrel{\widehat{=}} \\
&\quad \big[\, \Delta(\mathit{fleetP1})\ \mathit{fleetP1!}, \mathit{fleetP1'!} : \mathit{FLEETBOARD};\ \mathit{guess?} : \mathit{COORDINATE} \\
&\quad\ \mid \mathit{fleetP1'} = \mathit{fleetP1} \mathbin{⩤} \{\mathit{guess?}\} \land \\
&\quad\ \ \ \mathit{fleetP1!} = \mathit{fleetP1} \land \mathit{fleetP1'!} = \mathit{fleetP1'} \,\big] \parallel \\
&\quad \big[\, \Delta(\mathit{guessesP2})\ \mathit{fleetP1?}, \mathit{fleetP1'?} : \mathit{FLEETBOARD};\ \mathit{guess?} : \mathit{COORDINATE} \\
&\quad\ \mid \mathit{guess?} \notin \operatorname{dom} \mathit{guessesP2} \land \\
&\quad\ \ \ \mathit{guessesP2'} = \mathit{guessesP2} \cup \{\mathit{guess?} \mapsto \\
&\qquad\qquad \text{if } \mathit{fleetP1'?} \subset \mathit{fleetP1?} \text{ then} \\
&\qquad\qquad\quad \text{if } \operatorname{ran} \mathit{fleetP1'?} \subset \operatorname{ran} \mathit{fleetP1?} \text{ then} \\
&\qquad\qquad\qquad \text{if } \mathit{fleetP1'?} = \varnothing \text{ then} \\
&\qquad\qquad\qquad\quad \mathit{fleet\_destroyed} \text{ else } \mathit{sink} \text{ else } \mathit{hit} \text{ else } \mathit{miss}\} \,\big]
\end{aligned}
$$

We then separate and name the anonymous schemas so that they form distinct operations as required by the annealing rule's precondition. The particulars of the names chosen are arbitrary at this stage, as long as they are meaningful. In the case above, for example, we redefine *FireAtP*1 to be (*UpdateFleetP*1 ∥ *UpdateGuessesP*2) and define *UpdateFleetP*1 and *UpdateGuessesP*2 accordingly.

The annealing rule can then be applied to the *System* class, twice, producing a class for each player: *Player*1 is created (refer to Figure 4) and then *Player*2 (similar to *Player*1 but not shown for brevity). The schemas that apply to each player are migrated from *System* into each respective new class by the rule, and each class is instantiated once using the variable names *player*1 and *player*2. Dereferencing the *player*1 and *player*2 variables in *System* gives access to the migrated operations and variables. The state information and initialisation predicate for each player are also migrated, giving each class localised information pertaining to the player it represents.

Coalescence

The single *Player* class, shown in Figure 5, is formed by applying the coalescence rule to the *Player*1 and *Player*2 classes. The trivial application of the coalescence rule requires that the two classes be exactly the same, which in this case can be achieved through renaming attributes and operations (see Section 4) in each class to be those names shown in Figure 5. This circumvents the need to discharge proofs showing the *Player* class to be a refinement of both *Player*1 and *Player*2.

This completes the process of creating a separate class to encapsulate the functionality of a player. However, the placement of game logic responsible for player interaction still resides in the *System* class. Section 6 illustrates the process of enacting an alternate design decision: moving this logic into the new *Player* class.

6 Restructuring for Active Players

At this stage of the design process, a fairly significant design decision has already been made. The *Player* class formed by annealing and coalescing from the *System* class creates *Player* objects that are *passive*. That is, the logic of the

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{System}} \\ \hline
\upharpoonright(\textsc{Init}, \textit{InitFleetP}1, \textit{InitFleetP}2, \textit{FireAtP}1, \textit{FireAtP}2) \\[4pt]
\begin{array}{|l}
\hline
\textit{guessesP}2 : \textit{GUESSBOARD} \\
\textit{fleetP}2 : \textit{FLEETBOARD} \\
\textit{player}1 : \textit{Player}1 \\ \hline
\textit{player}1.\textit{system} = \textit{self} \\ \hline
\end{array}
\quad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitFleetP}2} \\ \hline
\Delta(\textit{fleetP}2) \\
\textit{fleet}? : \textit{FLEETBOARD} \\ \hline
\textit{fleetP}2' = \textit{fleet}? \\ \hline
\end{array} \\[4pt]
\begin{array}{|l}
\multicolumn{1}{l}{\textsc{Init}} \\ \hline
\textit{guessesP}2 = \varnothing \\
\textit{fleetP}2 = \varnothing \\
\textit{player}1.\textsc{Init} \\ \hline
\end{array} \\[4pt]
\textit{InitFleetP}1 \mathrel{\widehat=} \textit{player}1.\textit{InitFleetP}1 \\
\textit{FireAtP}1 \mathrel{\widehat=} \textit{player}1.\textit{UpdateFleetP}1 \parallel \textit{UpdateGuessesP}2 \\
\textit{UpdateGuessesP}2 \mathrel{\widehat=} \\
\quad \big[\, \Delta(\textit{guessesP}2)\ \textit{fleetP}1?, \textit{fleetP}1'? : \textit{FLEETBOARD} \\
\qquad\qquad\qquad \textit{guess}? : \textit{COORDINATE} \\
\quad \mid \textit{guess}? \notin \operatorname{dom} \textit{guessesP}2 \land \textit{guessesP}2' = \textit{guessesP}2 \cup \{\textit{guess}? \mapsto \\
\qquad\qquad \text{if } \textit{fleetP}1'? \subset \textit{fleetP}1? \text{ then} \\
\qquad\qquad\quad \text{if } \operatorname{ran} \textit{fleetP}1'? \subset \operatorname{ran} \textit{fleetP}1? \text{ then} \\
\qquad\qquad\qquad \text{if } \textit{fleetP}1'? = \varnothing \text{ then} \\
\qquad\qquad\qquad\quad \textit{fleet\_destroyed} \text{ else } \textit{sink} \text{ else } \textit{hit} \text{ else } \textit{miss}\} \,\big] \\
\textit{FireAtP}2 \mathrel{\widehat=} \textit{UpdateFleetP}2 \parallel \textit{player}1.\textit{UpdateGuessesP}1 \\
\textit{UpdateFleetP}2 \mathrel{\widehat=} \\
\quad \big[\, \Delta(\textit{fleetP}2)\ \textit{fleetP}2!, \textit{fleetP}2'! : \textit{FLEETBOARD} \\
\qquad\qquad\qquad \textit{guess}? : \textit{COORDINATE} \\
\quad \mid \textit{fleetP}2' = \textit{fleetP}2 \mathbin{\lhd} \{\textit{guess}?\} \land \\
\qquad \textit{fleetP}2! = \textit{fleetP}2 \land \textit{fleetP}2'! = \textit{fleetP}2' \,\big] \\ \hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Player}1} \\ \hline
\upharpoonright(\textsc{Init}, \textit{system}, \textit{InitFleetP}1, \textit{UpdateFleetP}1, \textit{UpdateGuessesP}1) \\[4pt]
\begin{array}{|l}
\hline
\textit{guessesP}1 : \textit{GUESSBOARD} \\
\textit{fleetP}1 : \textit{FLEETBOARD} \\
\textit{system} : \textit{System} \\ \hline
\end{array}
\quad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitFleetP}1} \\ \hline
\Delta(\textit{fleetP}1) \\
\textit{fleet}? : \textit{FLEETBOARD} \\ \hline
\textit{fleetP}1' = \textit{fleet}? \\ \hline
\end{array} \\[4pt]
\begin{array}{|l}
\multicolumn{1}{l}{\textsc{Init}} \\ \hline
\textit{guessesP}1 = \varnothing \\
\textit{fleetP}1 = \varnothing \\ \hline
\end{array} \\[4pt]
\textit{UpdateFleetP}1 \mathrel{\widehat=} \big[\, \Delta(\textit{fleetP}1)\ \textit{fleetP}1!, \textit{fleetP}1'! : \textit{FLEETBOARD} \\
\qquad\qquad\qquad \mid \textit{fleetP}1' = \textit{fleetP}1 \mathbin{\lhd} \{\textit{guess}?\} \land \\
\qquad\qquad\qquad\quad \textit{fleetP}1! = \textit{fleetP}1 \land \textit{fleetP}1'! = \textit{fleetP}1' \,\big] \\
\textit{UpdateGuessesP}1 \mathrel{\widehat=} \\
\quad \big[\, \Delta(\textit{guessesP}1)\ \textit{fleetP}2?, \textit{fleetP}2'? : \textit{FLEETBOARD} \\
\qquad\qquad\qquad \textit{guess}? : \textit{COORDINATE} \\
\quad \mid \textit{guess}? \notin \operatorname{dom} \textit{guessesP}1 \land \textit{guessesP}1' = \textit{guessesP}1 \cup \{\textit{guess}? \mapsto \\
\qquad\qquad \text{if } \textit{fleetP}2'? \subset \textit{fleetP}2? \text{ then} \\
\qquad\qquad\quad \text{if } \operatorname{ran} \textit{fleetP}2'? \subset \operatorname{ran} \textit{fleetP}2? \text{ then} \\
\qquad\qquad\qquad \text{if } \textit{fleetP}2'? = \varnothing \text{ then} \\
\qquad\qquad\qquad\quad \textit{fleet\_destroyed} \text{ else } \textit{sink} \text{ else } \textit{hit} \text{ else } \textit{miss}\} \,\big] \\ \hline
\end{array}
$$

Fig. 4. After the annealing step for *Player*1.

```
System
  ↾(INIT, InitFleetP1, InitFleetP2, FireAtP1, FireAtP2)

  | player1, player2 : Player
  |----------------
  | player1.system = self
  | player2.system = self

  INIT
  | player1.INIT
  | player2.INIT

  InitFleetP1 ≙ player1.InitFleet
  InitFleetP2 ≙ player2.InitFleet
  FireAtP1 ≙ player1.UpdateFleet ∥ player2.UpdateGuesses
  FireAtP2 ≙ player2.UpdateFleet ∥ player1.UpdateGuesses

Player
  ↾(INIT, system, InitFleet, UpdateFleet, UpdateGuesses)

  | guesses : GUESSBOARD
  | fleet : FLEETBOARD
  | system : System

  INIT
  | guesses = ∅
  | fleet = ∅

  UpdateFleet
  | Δ(fleet)
  | fleet!, fleet′! : FLEETBOARD
  | guess? : COORDINATE
  |----------------
  | fleet′ = fleet ⩤ {guess?}
  | fleet! = fleet ∧ fleet′! = fleet′

  UpdateGuesses
  | Δ(guesses)
  | fleet?, fleet′? : FLEETBOARD
  | guess? : COORDINATE
  |----------------
  | guess? ∉ dom guesses
  | guesses′ = guesses ∪ {guess? ↦
  |   if fleet′? ⊂ fleet? then
  |     if ran fleet′? ⊂ ran fleet? then
  |       if fleet′? = ∅ then
  |         fleet_destroyed
  |       else sink else hit else miss}

  InitFleet
  | Δ(fleet)
  | fleet? : FLEETBOARD
  |----------------
  | fleet′ = fleet?
```

Fig. 5. After the coalescence step.

interactions between the players of the system is contained within the *System* class and not the *Player* classes themselves (the *System* acts as a mediator). An alternative design moves this logic into the *Player* class, such that each player instance holds a reference directly to the other player and the interaction functionality is implemented by bypassing the *System* class altogether.

Even this solution has two possible modes: each player could either have a *Fire* operation that launches a missile at the opponent, or a *Defend* operation that causes the opponent to fire a missile at the player object invoking the operation. All solutions are consistent with the original specification, but having active *Player* objects with a *Fire* operation is more intuitive, from a user-interface perspective, than the passive approach presented earlier or having a *Defend* operation.

To refactor *Fire* into the design, it is necessary that we go back to just after the two annealing steps applied in Section 5, and:

Step 1. Move the body of the $\mathit{FireAtP}1$ operation into $\mathit{Player}2$ and name it Fire. Retain the operation $\mathit{FireAtP}1$ because it appears in the interface of System, but redefine it as $\mathit{player}2.\mathit{Fire}$. We repeat this process accordingly to move the $\mathit{FireAtP}2$ operation into the $\mathit{Player}1$ class.

For example, for class $\mathit{Player}1$:

$$\mathit{Fire} \mathrel{\widehat{=}} \mathit{system}.\mathit{player}2.\mathit{UpdateFleet} \parallel \mathit{system}.\mathit{player}1.\mathit{UpdateGuesses}$$

Step 2. We simplify the backward references to system by noting that the $\mathit{Player}1$ and $\mathit{Player}2$ classes are only ever instantiated once. Therefore self can replace $\mathit{system}.\mathit{player}1$ in $\mathit{Player}1$, and self can replace $\mathit{system}.\mathit{player}2$ in $\mathit{Player}2$.

Step 3. We introduce a local $\mathit{opponent}$ variable to invariantly equal $\mathit{system}.\mathit{player}2$ in $\mathit{Player}1$. Similarly for $\mathit{Player}2$, $\mathit{opponent}$ is introduced to invariantly equal $\mathit{system}.\mathit{player}1$. This supplies a simple abbreviation, so showing the refinement is trivial.

Now, for both classes:

$$\mathit{Fire} \mathrel{\widehat{=}} \mathit{opponent}.\mathit{UpdateFleet} \parallel \mathit{self}.\mathit{UpdateGuesses}$$

With the invariants added:

$$\begin{array}{l}\text{For class } \mathit{Player}1\text{: } \mathit{opponent} = \mathit{system}.\mathit{player}2 \\ \text{For class } \mathit{Player}2\text{: } \mathit{opponent} = \mathit{system}.\mathit{player}1\end{array}$$

Step 4. We move these new invariant conditions into the System class for all instances of $\mathit{Player}1$ and $\mathit{Player}2$ (there is only one of each), and extend the interface of both classes $\mathit{Player}1$ and $\mathit{Player}2$ to include $\mathit{opponent}$.

Thus, the System class now contains the invariants:

$$\begin{array}{l}\mathit{player}1.\mathit{opponent} = \mathit{player}2 \\ \mathit{player}2.\mathit{opponent} = \mathit{player}1\end{array}$$

Step 5. In the same manner as described in Section 5, we apply the coalescence rule trivially by renaming operations and variables in the two player classes to make them correspond.

The changes to the specification from applying this process are illustrated in Figure 6. The resultant specification represents an improvement, in terms of design, over the original specification shown in Figure 1. This is evidenced by the reduction of redundancy by encapsulating the concept of a player inside a separate class. In addition, the design decision to move the interaction logic into the *Player* class is a step toward a distributed architecture, and also toward the provision of a separate user-interface for each *Player*.

System

$\upharpoonright(\textsc{Init}, InitFleetP1, InitFleetP2, FireAtP1, FireAtP2, player1, player2)$

$player1, player2 : Player$

$player1.system = self$
$player2.system = self$
$player1.opponent = player2$
$player2.opponent = player1$

Init

$player1.\textsc{Init}$
$player2.\textsc{Init}$

$InitFleetP1 \mathrel{\widehat=} player1.InitFleet$
$InitFleetP2 \mathrel{\widehat=} player2.InitFleet$
$FireAtP1 \mathrel{\widehat=} player2.Fire$
$FireAtP2 \mathrel{\widehat=} player1.Fire$

Player

$\upharpoonright(\textsc{Init}, system, InitFleet, Fire, opponent)$

$guesses : GUESSBOARD$
$fleet : FLEETBOARD$
$system : System$
$opponent : Player$

Init

$guesses = \varnothing$
$fleet = \varnothing$

InitFleet

$\Delta(fleet)$
$fleet? : FLEETBOARD$

$fleet' = fleet?$

UpdateFleet

$\Delta(fleet)$
$fleet!, fleet'! : FLEETBOARD$
$guess? : COORDINATE$

$fleet' = fleet \ntriangleleft \{guess?\}$
$fleet! = fleet \land fleet'! = fleet'$

UpdateGuesses

$\Delta(guesses)$
$fleet?, fleet'? : FLEETBOARD$
$guess? : COORDINATE$

```
$guess? \notin \mathrm{dom}\, guesses$
$guesses' = guesses \cup \{guess? \mapsto$
    if $fleet'? \subset fleet?$ then
       if ran $fleet'? \subset$ ran $fleet?$ then
          if $fleet'? = \varnothing$ then
            $fleet\_destroyed$
          else $sink$
       else $hit$
    else $miss\}]$
```

$Fire \mathrel{\widehat=} opponent.UpdateFleet \parallel self.UpdateGuesses$

Fig. 6. After the coalescence step, with active players.

7 Conclusions and Future Work

The presented transformation rules are intended to be general enough to allow for the introduction of a variety of different designs and architectures. It is recognised, however, that adding a rule to repeat structure, rather than just divide or remove structure, could lend itself to many more architecture types (particularly in distributed domains). Other rules for modifying inheritance relationships between classes would be useful, particularly when targeting languages that do not support multiple inheritance (such as Java), and is left as an area for future work.

The provision of tool support for automating the refactoring process has been achieved in the past [16,19] and is recognised as important. Tool support for the rules presented above is possible, and has been prototyped by formalising the rules in the Z specification language as operations upon a state containing the Object-Z specification (a meta-model). The rules are written as schemas that are executable by a specification animator, so the user may interact with the animation software to automate the refactoring process. The evolution of the case study presented in this paper has been successfully automated using this technique, taking advantage of the Possum [8] animator. The possibility exists for abstraction from the specification language itself to allow succinct formal specification and reasoning about these refactoring steps, in a similar vein to the work presented by Mens et al. [14] and Lämmel [20], and this is also left for future work.

The methodology is also intended to allow for an heuristic-based description of object-oriented design patterns as sequences of the refactoring steps. Some work has been performed toward this goal by introducing the Model-View-Controller architectural paradigm [9] for user interfaces into the above case study.

Acknowledgements

The author wishes to acknowledge Graeme Smith and Colin Fidge for their valuable input.

References

1. M. Bonsangue, J. Kok, and K. Sere. An approach to object–orientation in action systems. In *Mathematics of Program Construction (MPC'98)*, volume 1422 of *LNCS*, pages 68–95. Springer-Verlag, June 1998.
2. M. Bonsangue, J. Kok, and K. Sere. Developing object-based distributed systems. In *Proc. of the 3rd IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'99)*, pages 19–34. Kluwer, 1999.
3. E. Casais. An incremental class reorganization approach. In O. Madsen, editor, *Proc. ECOOP'92*, volume 615 of *LNCS*, pages 114–132. Springer-Verlag, June 1992.
4. M. Cornelio, A. Cavalcanti, and A. Sampaio. Refactoring by Transformation. In *Proc. of REFINE'2002*, Electronic Notes in Theoretical Computer Science. Elsevier, July 2002.
5. J. Derrick and E. Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications.* FACIT Series. Springer, May 2001.

6. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison–Wesley, 1999.
7. S. Goldsack and K. Lano. Annealing and data decomposition in VDM. *ACM Sigplan Notices*, 31(4):32–38, April 1996.
8. D. Hazel, P. Strooper, and O. Traynor. Possum: An animator for the SUM specification language. In W. Wong and K. Leung, editors, *Asia Pacific Software Engineering Conference (APSEC 97)*, pages 42–51. IEEE Computer Society, 1997.
9. T. Hopkins and B. Horan. *Smalltalk: An Introduction to Application Development Using VisualWorks*. Prentice-Hall, 1995.
10. K. Lano and S. Goldsack. Refinement of distributed object systems. In *Proc. of Workshop on Formal Methods for Open Object-based Distributed Systems*. Chapman and Hall, 1996.
11. K. Lano. *Formal Object-oriented Development*. Springer Verlag, 1995.
12. J. Lu. Introducing data decomposition into VDM for tractable development of programs. *ACM SIGPLAN Notices*, 30(9), September 1995.
13. Z. Manna and R. Waldinger. Synthesis: Dreams ⇒ programs. *IEEE Transactions on Software Engineering*, 5(4):294–328, July 1979.
14. T. Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In *Proc. Int'l Conf. Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag, 2002.
15. A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object–oriented programs. In *Proc. of the Fourth International Formal Methods Europe Symposium (FME'97)*, volume 1313 of *LNCS*. Springer-Verlag, 1997.
16. I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *OOPSLA '96: Proc. Object-Oriented Programming Systems, Languages, and Applications*, pages 235–250. ACM Press, 1996.
17. C. C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science, 2nd edition, 1994.
18. W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Computer Science Department, Urbana-Champaign, IL, USA, May 1992.
19. D. Roberts, J. Brant, and R. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3:253–263, 1997.
20. R. Lämmel. Towards generic refactoring. In *Third ACM SIGPLAN Workshop on Rule-Based Programming*. ACM Press, 2002.
21. G. Smith. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.
22. G. Smith. *The Object-Z Specification Language*. Kluwer, 2000.

Checking Absence of Illicit Applet Interactions: A Case Study*

Marieke Huisman[1], Dilian Gurov[2],
Christoph Sprenger[1], and Gennady Chugunov[3]

[1] INRIA Sophia Antipolis, France
[2] Royal Institute of Technology, Kista, Sweden
[3] Swedish Institute of Computer Science, Kista, Sweden

Abstract. This paper presents the use of a method – and its corresponding tool set – for compositional verification of applet interactions on a realistic industrial smart card case study. The case study, an electronic purse, is provided by smart card producer Gemplus as a test case for formal methods for smart cards. The verification method focuses on the possible interactions between different applets, co-existing on the same card, and provides a technique to specify and detect illicit interactions between these applets. The method is compositional, thus supporting post-issuance loading of applets. The correctness of a global system property can algorithmically be inferred from local applet properties. Later, when loading applets on a card, the implementations are matched against these local properties, in order to guarantee the global property. The theoretical framework underlying our method has been presented elsewhere; the present paper evaluates its practical usability by means of an industrial case study. In particular, we outline the tool set that we have assembled to support the verification process, combining existing model checkers with newly developed tools, tailored to our method.

1 Introduction

The growing market for smart cards and other small personal devices has increased the need to use formal validation and verification techniques in industry. These devices often contain privacy–sensitive information; this is the case in typical usages for smart cards such as health care information systems and electronic purses. Therefore strong security guarantees are needed for their wide–spread acceptance. With the acceptance of evaluation schemes such as Common Criteria[1] industry has come to realise that the only way to achieve such high guarantees is to adopt the use of formal methods in industrial practice.

Various work has been done, aiming at the verification of different kinds of properties of smart card applications. Properties under study are for example functional correctness, confidentiality, availability and restrictions on information flow. Often this work focuses on the correctness of a single applet, or of a set

* Partially supported by the EU as part of the VerifiCard project IST-2000-26328.
[1] See `http://www.commoncriteria.org`.

of applets that is known in advance. However, future generations of smart cards are expected to allow post–issuance loading of applets, where newly installed applets interact with the applets already present on the card. As a consequence, at the time the card is issued, it is not known which applets it might contain. Therefore, it is necessary to state minimal requirements for the applets that can be loaded later on the card, and to be able to verify at loading time that the applets actually respect these requirements. Only then, existing applets can safely communicate with new applets, without corrupting the security of the card.

In the present case study we focus on a particular kind of properties to ensure the security of the card, namely the absence of illicit control flow between the different applets. For multi–application smart cards, certain control flow paths can be undesirable because of general platform–dependent restrictions, like the recommendation to avoid recursion due to limited resources, or due to application–specific restrictions, like undesirable information flow caused by illicit applet interactions as studied in this paper.

In a companion paper we presented an algorithmic compositional verification technique for such control flow based safety properties [14], using a temporal logic specification language for specifying applet properties. These can be either structural, interpreting formulae over the control flow graph of an applet, or behavioural, interpreting formulae over applet behaviour. The approach is compositional in that it allows global control flow properties of the whole system to be inferred from local control flow properties of the individual applets. In this way, global security properties can be guaranteed to hold even in the presence of post–issuance loading of applets, as long as these applets satisfy their local properties. The latter check can be delegated to a separate authority not necessarily possessing the code of the applets already residing on the card. However, while the global properties can be behavioural or structural, we require the local properties to be structural; our technique does not allow global behavioural properties to be algorithmically inferred from local behavioural ones. For a more detailed motivation for using structural assumptions the reader is referred to [14].

An important asset of our method is that the verification tasks involved are all based on algorithmic techniques, as opposed to earlier work in which we developed a proof system for compositional verification [1]. Therefore, once the specifications for the different applets and the illicit applet interaction are given, all verifications can be done automatically, using push–button technology. This paper presents the tool set that we have assembled to support the whole verification process, and illustrates its usefulness by applying it to a realistic, industrial electronic purse case study, provided by the smart card producer Gemplus. The application is not actually used by Gemplus, but has been provided as a test case to apply formal methods to smart card applications. The properties that we verify illustrate typical application–dependent illicit applet interactions.

As far as we are aware, this work is the first to develop algorithmic techniques for the compositional verification of control flow properties for applets. Earlier, we used part of our tool set for non-compositional verification of control flow properties [8]. The underlying program model has been inspired by the work of

Besson *et al.* [2], who verify stack properties for Java programs. Our work differs considerably from more known model checkers for multi-threaded Java such as Bandera [11] and Java PathFinder [5]. In contrast to these tools, we focus on the control flow of applications and the compositionality of the verification. Finally, we mention the model checking algorithms for Push–Down Automata, developed by Bouajjani *et al.* [4]. We use the implementation of these algorithms in the model checker Alfred [13] to verify the correctness of the decomposition.

The paper is structured as follows. First, Section 2 outlines the general structure of the tool set. Next, Section 3 summarises the theoretical framework underlying our approach. Then, Section 4 introduces the electronic purse example, and motivates the property that we are interested in. This property is formalised in Section 5, together with appropriate local properties for the individual applets. Finally, Section 6 discusses the use of our tool set to establish the correctness of the property decomposition and of the local properties *w.r.t.* an implementation. For a more detailed account of the theoretical framework we refer to our companion paper [14].

2 General Overview of the Approach

As explained above, we aim at checking the absence of illicit applet interactions, given the possibility of post–issuance loading, by using a compositional verification method. In our method, we identify the following tasks:

1. specification of global security properties as behavioural safety properties;
2. specification of local properties as structural safety properties;
3. algorithmic verification of property decompositions, ensuring that the local properties imply the global ones; and
4. algorithmic verification of local properties for individual applets.

Our method is based on the construction of maximal applets *w.r.t.* structural safety properties. An applet is considered to be maximal *w.r.t.* a property if it simulates all applets respecting this property.

Concretely, suppose we want to prove that the composition of applets A and B respects a security property, formulated as behavioural safety property ϕ (Task 1). We specify structural properties σ_A and σ_B (Task 2) for which we construct maximal applets $\theta_{I_A}(\sigma_A)$ and $\theta_{I_B}(\sigma_B)$, respectively (where I_A and I_B are the interfaces of the applets A and B, respectively). We show, using existing model checking techniques, that their composition respects the behavioural safety property ϕ, *i.e.* $\theta_{I_A}(\sigma_A) \uplus \theta_{I_B}(\sigma_B) \models \phi$. The validity of this assertion corresponds to the correctness of the property decomposition (Task 3), since the simulation pre–order is preserved under applet composition and behavioural properties expressible in our logic are preserved by simulation. When we get concrete implementations for A and B, we use existing model checking techniques to check whether these implementations respect σ_A and σ_B, respectively (Task 4).

To support our compositional verification method, we have developed a tool set, combining existing model checking tools and newly developed tools, specific

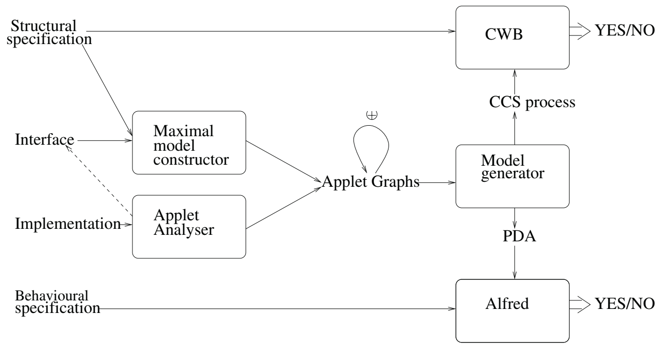


Fig. 1. Overview of tool set

to our method. Figure 1 gives a general overview of our tool set. Section 3 below introduces the underlying theoretical framework.

As input we have for each applet either an implementation, or a structural property, restricting its possible implementations, plus an interface, specifying the methods provided and required by the applet. For these inputs, we construct an applet representation, which is basically a collection of control flow graphs representing methods, plus the applet interface. In case we have the applet implementation, we use the *Applet Analyser* to extract the applet graph. In case we have a structural property, we use the *Maximal Model Constructor* to construct an applet graph that simulates all possible implementations of applets respecting the formula. For a given applet implementation, the Applet Analyser can also be used to obtain the applet interface. If required, applets can be composed, using the *applet composition* operator $\uplus$. This operation essentially corresponds to forming the disjoint union of applets. Using the *Model Generator* the resulting applet graphs are translated into models which serve as input for different model checkers. If we want to check structural properties, we translate the resulting graphs into CCS processes, which can be used as input for the Edinburgh Concurrency Workbench (CWB) [9]. If for a composed system we want to verify whether it respects a behavioural safety property, we translate the composed graphs into Push–Down Automata (PDA), which form the input for the model checker Alfred [13].

3 A Framework for Compositional Verification

This section outlines the theoretical framework underlying our tool set. For a more comprehensive account of the technical details the reader is referred to [14].

3.1 Program Model

As we are only studying control flow properties, we abstract away from all data in our program model. Further, since we are only concerned with smart card applications, we only consider sequential programs[2]. Basically, an applet is a collection of method graphs, plus an interface specifying which methods it provides and requires. For each method, there is a method graph describing its possible control flow. Edges in the graphs denote method calls or internal computations.

As explained above, we distinguish between structural level properties, restricting possible implementations of methods, and behavioural level properties, restricting the possible behaviour of methods. Therefore, we also have two different views on applets (and methods): structural and behavioural. However, these two views are instantiations of a single framework (see [14]).

General Framework. First we present the general framework, defining the notions of *model* and *specification* over a set of labels L and a set of atomic propositions A. These are later instantiated to the structural and behavioural level.

Definition 1. (Model) *A* model *over labels L and atomic propositions A is a structure $\mathcal{M} = (S, L, \rightarrow, A, \lambda)$, where S is a set of states, L is a finite set of labels, $\rightarrow \subseteq S \times L \times S$ is a transition relation, A is a finite set of atomic propositions, and $\lambda : S \rightarrow \mathcal{P}(A)$ is a valuation assigning to each state s the atomic propositions that hold at s. A* specification *$\mathcal{S}$ over L and A is a pair $(\mathcal{M}, E)$, where $\mathcal{M}$ is a model over L and A and $E \subseteq S$ is a set of states.*

Intuitively, one can think of E as the set of entry states of the model. We define the usual notion of simulation $\leq$ (where related states satisfy the same atomic propositions).

Applet Structure. Before instantiating the notion of model on the structural level, we first define the notion of applet interface. Let $\mathcal{M}eth$ be a countably infinite set of method names.

Definition 2. (Applet interface) *An* applet interface *is a pair $I = (I^+, I^-)$, where $I^+, I^- \subseteq \mathcal{M}eth$ are finite sets of names of* provided *and* required *methods, respectively. The* composition *of two interfaces $I_1 = (I_1^+, I_1^-)$ and $I_2 = (I_2^+, I_2^-)$ is defined by $I_1 \cup I_2 = (I_1^+ \cup I_2^+, I_1^- \cup I_2^-)$.*

As mentioned above, a method specification is an instance of the general notion of specification.

Definition 3. (Method specification) *A* method graph *for $m \in \mathcal{M}eth$ over a set M of method names is a finite model*

$$\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$$

where V_m is the set of control nodes of m, $L_m = M \cup \{\varepsilon\}$, $A_m = \{m, r\}$, $m \in \lambda_m(v)$ for all $v \in V_m$, i.e. *each node is tagged with the method name, and the nodes $v \in V_m$ with $r \in \lambda_m(v)$ are* return points*. A* method specification *for $m \in \mathcal{M}eth$ over M is a pair $(\mathcal{M}_m, E_m)$, where $\mathcal{M}_m$ is a method graph for m over M and $E_m \subseteq V_m$ is a non–empty set of* entry points *of m.*

[2] For example, Java Card, a dialect of Java for programming smart cards, does currently not allow multi-threading.

Table 1. Applet Transition Rules

$$\text{(transfer)} \quad \frac{m \in I^+ \qquad v \to_m v' \qquad v \models \neg r}{(v, \sigma) \xrightarrow{\varepsilon} (v', \sigma)}$$

$$\text{(call)} \quad \frac{m_1, m_2 \in I^+ \qquad v_1 \xrightarrow{m_2}_{m_1} v_1' \qquad v_1 \models \neg r \qquad v_2 \models m_2 \qquad v_2 \in E}{(v_1, \sigma) \xrightarrow{m_1 \mathsf{call}\ m_2} (v_2, v_1' \cdot \sigma)}$$

$$\text{(return)} \quad \frac{m_1, m_2 \in I^+ \qquad v_2 \models m_2 \wedge r \qquad v_1 \models m_1}{(v_2, v_1 \cdot \sigma) \xrightarrow{m_2\ \mathsf{ret}\ m_1} (v_1, \sigma)}$$

An applet is basically a collection of method specifications and an interface. For the formal definition we use the notion of disjoint union of specifications $\mathcal{S}_1 \uplus \mathcal{S}_2$, where each state is tagged with 1 or 2, respectively, and $(s, i) \xrightarrow{a}_{\mathcal{S}_1 \uplus \mathcal{S}_2} (t, i)$, for $i \in \{1, 2\}$, if and only if $s \xrightarrow{a}_{\mathcal{S}_i} t$.

Definition 4. (Applet) *An* applet $\mathcal{A}$ *with interface* I*, written* $\mathcal{A} : I$*, is defined inductively by*

– $(\mathcal{M}_m, E_m) : (\{m\}, M)$ *if* $(\mathcal{M}_m, E_m)$ *is a method specification for* $m \in \mathcal{M}eth$ *over* M*, and*
– $\mathcal{A}_1 \uplus \mathcal{A}_2 : I_1 \cup I_2$ *if* $\mathcal{A}_1 : I_1$ *and* $\mathcal{A}_2 : I_2$*.*

An applet is *closed* if $I^- \subseteq I^+$, *i.e.* it does not require any external methods. Simulation instantiated to this particular type of models is called structural simulation, denoted as $\leq_s$.

Applet Behaviour. Next we instantiate specifications on the behavioural level.

Definition 5. (Behaviour) *Let* $\mathcal{A} = (\mathcal{M}, E) : (I^+, I^-)$ *be a closed applet where* $\mathcal{M} = (V, L, \to, A, \lambda)$*. The* behaviour *of* $\mathcal{A}$ *is described by the specification* $b(\mathcal{A}) = (\mathcal{M}_b, E_b)$*, where* $\mathcal{M}_b = (S_b, L_b, \to_b, A_b, \lambda_b)$ *such that* $S_b = V \times V^*$, i.e. *states are pairs of control points and stacks,* $L_b = \{m_1\ l\ m_2 \mid l \in \{\mathsf{call}, \mathsf{ret}\}, m_1, m_2 \in I^+\} \cup \{\varepsilon\}$*,* $\to_b$ *is defined by the rules of Table 1,* $A_b = A$*, and* $\lambda_b((v, \sigma)) = \lambda(v)$*.*

The set of initial states E_b *is defined by* $E_b = E \times \{\varepsilon\}$*, where* ε *denotes the empty sequence over* V*.*

Note that applet behaviour defines a Push–Down Automaton (see, *e.g.*, [7] for a survey of verification techniques for infinite process structures). We exploit this by using a model checker for PDAs to verify behavioural properties.

Also on the behavioural level, we instantiate the definition of simulation $\leq_b$. Any two applets that are related by structural simulation, are also related by behavioural simulation, but the converse is not true (since behavioural simulation only requires reachable states to be related).

3.2 Property Specification Language

We use a fragment of the modal μ–calculus [12], namely the one excluding diamonds and least fixed points, to express properties restricting applet structure and behaviour[3]. We call this fragment *simulation logic*, because it is able to characterise simulation logically and, vice versa, satisfaction of a formula corresponds to being simulated by a maximal model derived from the formula. Similar logics have been studied earlier for capturing branching–time safety properties (see e.g. [3]). Let $\mathcal{X}$ be a countably infinite set of variables over sets of states. Let $X \in \mathcal{X}$, $a \in L$ and $p \in A$ denote state variables, labels and atomic propositions, respectively. The formulae in simulation logic are inductively defined as follows.

$$\phi ::= p \mid \neg p \mid X \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [a]\phi \mid \nu X.\phi$$

We only consider closed formulae of simulation logic, *i.e.* all variables $X \in \mathcal{X}$ have to be bound by some binder νX. Their semantics is standard, see *e.g.* Kozen [12]. The satisfaction relation is extended from states to specifications as usual: a specification satisfies a formula if all its entry points do. This relation is instantiated at both the structural and the behavioural level, denoted as $\models_s$ and $\models_b$, respectively. For each applet $\mathcal{A} : I$, we have an atomic proposition for each $m \in I^+$ and an atomic proposition r. At the structural level, labels are in $I^- \cup \{\epsilon\}$, and boxes are interpreted over edges in the method graphs. At the behavioural level, labels are in L_b (see Definition 5), and boxes are interpreted over transitions (see Table 1).

Writing specifications in the modal μ–calculus is known to be hard (even in our fragment), therefore we define a collection of commonly used specification patterns (inspired by the Bandera Specification Pattern project [10]). In our experience, all relevant behavioural control flow safety properties can be expressed using a small set of such patterns – however, it is important to remember that one can always fall back on the full expressiveness of simulation logic. Below we present several specification patterns, both at structural and behavioural level. These are all used in the case study at hand.

Structural Specification Patterns. We shall use *Everywhere* with the obvious formalisation:

$$\mathit{Everywhere}\,\sigma \;=\; \nu Z.\, \sigma \wedge [\varepsilon, I^-]Z$$

as well as the following patterns, for method sets M and M' of an applet with interface I:

$$\begin{aligned} M \ \mathit{HasNoCallsTo}\ M' &= \left(\textstyle\bigwedge_{m \in M} \neg m\right) \vee (\mathit{Everywhere}\ [M']\,\mathsf{false}) \\ \mathit{HasNoOutsideCalls}\ M &= M\ \mathit{HasNoCallsTo}\ (I^- \setminus M) \end{aligned}$$

The first pattern specifies that method graphs in the set M do not contain edges labelled with elements of the set M'. The second specifies a closed set of methods M, *i.e.* methods in M only contain calls to methods in M.

[3] In fact, in our theoretical framework, we use an alternative, but equivalent formulation, expressing formulae as modal equation systems.

Behavioural Specification Patterns. Pattern *Always* is standard:

$$\mathit{Always}\,\phi = \nu Z.\, \phi \wedge [L_b]Z$$

For specifying that a property ϕ is to hold within a call to method m, we use the *Within* pattern formalised as follows:

$$\mathit{Within}\, m\, \phi \;\; = \;\; \neg m \vee (\mathit{Always}\,\phi)$$

Notice that this is a typical behavioural pattern: the notion of *Within* a method invocation encompasses all methods that might be invoked during the call to m. This reachability notion cannot directly be expressed at the structural level.

Finally, for applet $\mathcal{A} : (I^+, I^-)$ and method set M, we define:

$$\mathit{CanNotCall}\,\mathcal{A}\; M \;\; = \;\; \bigwedge_{m \in I^+} \; \bigwedge_{m' \in M} [m\, \mathit{call}\, m']\, \mathsf{false}$$

This pattern holds for state (v, σ) if no call to a method in M is possible.

3.3 Maximal Models and Compositional Verification

Our compositional verification rests on the idea of constructing a so–called maximal model for a given property (*w.r.t.* a simulation pre–order). For every structural property σ and applet interface I, we can construct a so–called *maximal applet* $\theta_I(\sigma)$, *i.e.* an applet with interface I that simulates all applets with this interface, respecting property σ. As the simulation pre–order is preserved under applet composition and behavioural properties expressible in the logic are preserved by the simulation pre–order, we have the following compositional verification principle:

$$\frac{\mathcal{A} \models_s \sigma \qquad \theta_I(\sigma) \uplus \mathcal{B} \models_b \phi}{\mathcal{A} \uplus \mathcal{B} \models_b \phi}\ (\mathsf{beh\text{-}comp})$$

This rule states that the composition of applets $\mathcal{A}\colon I$ and $\mathcal{B}\colon J$ satisfies (global) behavioural property ϕ, if one can find a (local) structural property σ, satisfied by $\mathcal{A}$, such that the composition of the maximal applet *w.r.t.* σ and interface I, composed with applet $\mathcal{B}$ satisfies property ϕ. Thus, if we are given a structural property for an applet $\mathcal{A}$ and an implementation for an applet $\mathcal{B}$ we can verify whether their composition satisfies the required properties. We use the Maximal Model Constructor to compute $\theta_I(\sigma)$, the Applet Analyser to extract the applet graph for $\mathcal{B}$, and the Model Generator to produce input for Alfred, so it can check $\theta_I(\sigma) \uplus \mathcal{B} \models_b \phi$. Later, when an implementation for applet $\mathcal{A}$ becomes available, it can be verified independently whether it respects σ, by using the Applet Analyser to extract the applet graph for $\mathcal{A}$, and the Model Generator to generate input for CWB, which is used to check structural properties.

Note that, since applet composition is commutative, we can apply the composition principle above to its second premise and also replace applet $\mathcal{B}$ by a local structural property (in the same way as displayed above for applet $\mathcal{A}$).

4 Illicit Applet Interactions in the Electronic Purse

The Gemplus electronic purse case study PACAP [6] is developed to provide a realistic case study for applying formal methods to Java Card applications. The case study defines three applications: *CardIssuer*, *Purse* and *Loyalty*. Typically, a card will contain one card issuer and one purse applet, but several loyalty applets. The property that we verify for this case study only is concerned with *Purse* and *Loyalty,* therefore we will not discuss *CardIssuer* any further. If the card holder wishes to join a loyalty program, the appropriate applet can be loaded on the card. Subsequently, the purse and the different loyalties will exchange information about the purchases made, so the loyalty points can be credited. Current versions of Java Card use shareable interfaces to exchange this kind of information, but in the future this is likely to change. However, for our techniques it is not relevant how this communication exactly takes place, we only require that it is done in terms of method calls. The goal of our work is to ensure that no illicit interactions can happen between the applets on the card.

To understand the property that we are interested in, we look closer at how the purse and the loyalties communicate about the purchases made with the card. For efficiency reasons, the electronic purse keeps a log table of all credit and debit transactions, and the loyalty applets can request the (relevant) information stored in this table. Further, loyalties might have so–called partner loyalties, which means that a user can add up the points obtained with the different loyalty programs. Therefore, each loyalty should keep track of its balance and a so–called extended balance. If the user wishes to know how many loyalty points are available exactly, the loyalty applet will ask for the relevant entries of the purse's log table in order to update its balance, and it will also ask the balances of partner loyalties in order to compute the extended balance.

If the log table is full, existing entries will be replaced by new transactions. In order to ensure that loyalties do not miss any of the logged transactions, they can subscribe to the so–called *logFull* service. This service signals all subscribed loyalties that the log table will be overwritten soon, and that therefore they should update their balances. Typically, loyalties will have to pay for this service.

Suppose we have an electronic purse, which contains besides the electronic purse itself two partner loyalties, say L_1 and L_2. Further, suppose that L_1 has subscribed to the *logFull* service, while L_2 has not. If in reaction to the *logFull* message L_1 always calls an interface method of L_2 (say to ask for its balance), L_2 can implicitly deduce that the log table might be full. A malicious implementation of L_2 might therefore request the information stored in the log table before returning the value of its local balance to L_1. If loyalties have to pay for the *logFull* service, such control flow is unwanted, since the owner of the *Purse* applet will not want other loyalties to get this information for free.

This is a typical example of an illicit applet interaction, that our compositional verification technique can detect. Below, we show how the absence of this particular undesired scenario can be specified and verified algorithmically. We allow an arbitrary number of loyalty applets on the card. Since all loyalty applets have the same interface, we apply class–based analysis. We assume that at verification time only the *Purse* applet has been loaded on the card; the code of the loyalty applet class is not yet available. We use compositional reasoning to reduce the global behavioural property expressing the absence of the scenario described above to local structural properties of the purse and loyalty applet classes. The purse applet code is then checked against its structural property. When the loyalty applet code becomes available, possibly after the card has been issued, it is checked against its structural property before loading it on the card.

5 Specification

This section presents the formalisation of the global and local security properties that we need for our example. The next section discusses the verification of the decomposition and of the implementations *w.r.t.* the local properties.

As mentioned above, communication between applets takes place via so–called shareable interfaces. The *Purse* applet defines a shareable interface for communication with loyalty applets, containing among others the methods *getTransaction,* and *isThereTransaction.* The *Loyalty* applet defines shareable interfaces for communication with *Purse* and with other loyalty applets, containing among others the method *logFull.* The set I_P^+ denotes the methods provided by *Purse,* and M_L^{SI} denotes the set of shareable interface methods of *Loyalty.*

The Global Security Property. To guarantee that no loyalty will get the opportunity to circumvent subscribing to the *logFull* service, we require that if the *Purse* calls the *logFull* method of a loyalty, within this call the loyalty does not communicate with other loyalties. However, as the *logFull* method is supposed to call the *Purse* for its transactions, we also have to exclude indirect communications, via the *Purse.* We require the following global behavioural property:

A call to *Loyalty.logFull* does not trigger any calls to any other loyalty.

This property can be formalised with the help of behavioural patterns:

$$(\phi)\ \ \mathit{Within\ Loyalty.logFull}$$
$$(\mathit{CanNotCall\ Loyalty}\ M_L^{SI}) \wedge (\mathit{CanNotCall\ Purse}\ M_L^{SI})$$

Thus, if loyalty receives a *logFull* message, it cannot call any other loyalty (because it cannot call any of its shareable interface methods), and in addition, if the *Purse* is activated within the call to *logFull,* it cannot call any loyalty applet.

Property Decomposition. Next, we phrase local structural properties for *Purse* and *Loyalty*. Here we explain their formalisation; Section 6 presents how we actually verify that they are sufficient to guarantee the global behavioural property. Within *Loyalty.logFull,* the *Loyalty* applet has to call the methods *Purse.isThereTransaction* and *Purse.getTransaction*, but it should not make any other external calls (where calls to shareable interface methods of *Loyalty* are considered external[4]). Thus, a natural structural property for *Loyalty* would be, informally:

[4] Notice that since we are performing class–based analysis, we cannot distinguish between calls to interface methods of other instances, and those of the same instance.

> From any entry point of *Loyalty.logFull*, the only reachable external calls are calls to *Purse.isThereTransaction* and *Purse.getTransaction*.

Reachability is understood in terms of an extended graph of *Loyalty* containing explicit inter–method call edges.

For the *Purse* applet we know that within a call to *Loyalty.logFull* it can only be activated via *Purse.isThereTransaction* or *Purse.getTransaction*.

> From any entry point of *Purse.isThereTransaction* or *Purse.getTransaction*, no external call is reachable.

Again, reachability should be understood in terms of a graph containing explicit inter–method call edges. As our program model does not contain these, the above properties cannot be formalised directly in our logic. However, they can be formalised on a meta–level; for example for the *Purse,* the property holds, if and only if there exist sets of methods $M_{gT} \subseteq I_P^+$, containing *Purse.getTransaction*, and $M_{iTT} \subseteq I_P^+$, containing *Purse.isThereTransaction,* such that:

$$(\sigma_P)\ \mathit{HasNoOutsideCalls}\ M_{iTT}\ \wedge\ \mathit{HasNoOutsideCalls}\ M_{gT}$$

These sets represent the methods in *Purse* which can be called transitively from *Purse.isThereTransaction* and *Purse.getTransaction*, respectively. We can use the Applet Analyser to find them. Similarly, to express the property for *Loyalty* we need a set of methods $M_{lF} \subseteq I_L^+$ containing *Loyalty.logFull*, such that:

$$(\sigma_L)\ M_{lF}\ \mathit{HasNoCallsTo}\ I_L^- \setminus \left(M \setminus M_L^{SI}\right)$$

where $M = M_{lF} \cup \{Purse.isThereTransaction, Purse.getTransaction\}$. Calls to M_L^{SI} are excluded, since, as explained above, the methods in M_L^{SI} are treated as external. Since we assume that the code of the loyalty applet class is not yet available at verification time, M_{lF} has to be guessed. Here we take the (possibly too) simple choice $M_{lF} = \{Loyalty.logFull\}$. Under this choice, σ_L simplifies to $M_{lF}\ \mathit{HasNoCallsTo}\ I_L^- \setminus \{Purse.isThereTransaction, Purse.getTransaction\}$. However, if later one wishes to load an implementation of *Loyalty* with a different set M_{lF}, correctness of the decomposition can be re–established automatically.

6 Verification

Now that we have specified global and local security properties, we have to show: (1) the local properties are sufficient to establish the global security property, and (2) the implementations of the different applets respect the local properties. In order to do this, we identify the following (independent) tasks, discussed below.

1. Verifying the correctness of the property decomposition by:
   (a) building $\theta_{I_P}(\sigma_P)$ and $\theta_{I_L}(\sigma_L)$, the maximal applets for σ_P and σ_L; and
   (b) model checking $\theta_{I_P}(\sigma_P) \uplus \theta_{I_L}(\sigma_L) \models_b \phi$.
2. Verifying the local structural properties by:
   (a) extracting the applet graphs P of the *Purse* and L of the *Loyalty*; and
   (b) model checking $P \models_s \sigma_P$ and $L \models_s \sigma_L$.

Table 2. Statistics for maximal applet construction.

	# nodes	# edges	constr. time
σ_L	474	277 700	25 min.
σ_P	2 786	603 128	13 hrs.

As explained above, we have developed a tool set to support these verification tasks, combining existing model checking tools (CWB and Alfred) with our own tools (Maximal Model Constructor, Applet Analyser and the Model Generator).

6.1 Correctness of the Property Decomposition

To check correctness of the property decomposition, we construct maximal applets *w.r.t.* the specifications of the *Purse* and the *Loyalty*, and verify whether their composition respects the global behavioural property.

Constructing Maximal Applets. Given applet interface I and structural safety property σ, we produce $\theta_I(\sigma)$, the maximal applet for I and σ, using the procedure described in [14], implemented in Ocaml as the Maximal Model Constructor. The construction proceeds in three steps. First, the interface I is translated into a structural safety property characterising all behaviour possible under this interface. Then, the conjunction of this formula and the property σ is transformed into a semantically equivalent normal form, which can directly be translated into a model. This model is the maximal applet $\theta_I(\sigma)$. In general, the size of a maximal applet is exponential in the size of the input. We implemented some optimisations, which save both time and, more importantly, memory.

In the maximal applet for σ_L we can distinguish between two kinds of methods, which are illustrated in Figure 2: the methods in M_{lF} (that is *logFull)* have the left method graph, and only contain calls to *Purse.iTT* and *Purse.gT*. All other methods provided by *Loyalty* have the form of the right method graph, and do not contain any restrictions on the method calls. Each method of the applet $\theta_{I_L}(\sigma_L)$ has two nodes. The maximal applet for σ_P is similar, but each method consists of two to eight nodes depending on the set it belongs to (M_{iTT}, M_{gT} or I_P^+). Table 2 provides statistics on the size of the constructed graphs, and the corresponding construction time on a Pentium 1.9 GHz machine.

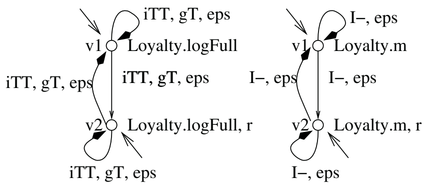


Fig. 2. Methods in $\theta_{I_L}(\sigma_L)$

Model Checking Behavioural Properties. Once the maximal applets $\theta_{I_P}(\sigma_P)$ and $\theta_{I_L}(\sigma_L)$ are constructed, we produce their composition $\theta_{I_P}(\sigma_P) \uplus \theta_{I_L}(\sigma_L)$. The behaviour of this applet is a (possibly infinite state) model generated by a pushdown automaton (PDA) given as a set of production rules. The model checking

Table 3. Statistics on applet graph extraction and verification.

	# classes	# methods	# nodes	# edges	extr. time	verif. time
Loyalty	11	237	3 782	4 372	5.6 sec.	12 sec.
Purse	15	367	5 882	7 205	7.5 sec.	19 sec.

problem for this class of models is exponential both in the size of the formula and in the number of control states of the PDA [7]. We base our experiments on Alfred [13], a demonstrator tool for model checking alternation–free modal μ–calculus properties of PDAs. We developed the Model Generator – implemented in Java – to translate applet graphs (in this case $\theta_{I_P}(\sigma_P) \uplus \theta_{I_L}(\sigma_L)$) to a PDA representation, which serves as input to Alfred. We were successful in checking correctness of (similar) property decompositions for applets with a small number of interface methods; when dealing with applets with large interfaces as in our case study, however, Alfred failed to scale up. Currently, we are investigating how to encode applets more efficiently, into context-free processes, which are equivalent to PDAs with a single control state. For this class of processes the model checking complexity becomes polynomial in the number of productions.

6.2 Correctness of the Local Structural Properties

Extracting Applet Graphs. The Applet Analyser is used to extract applet graphs and the appropriate set of entry points from the byte code of an applet. This is a static analysis tool, built on top of the SOOT Java Optimization Framework [15]. The byte code of a Java Card applet is transformed into Jimple basic blocks, while abstracting away variables, method parameters, and calls to methods of the Java Card API. We use SOOT's standard class hierarchy analysis to produce a safe over-approximation of the call graph. If, for example, the static analysis cannot determine the receiver of a virtual method call, a call edge is generated for every possible method implementation. Table 3 provides statistics on the extracted applet graphs.

Model Checking Structural Properties. Applet graphs can be viewed as finite Kripke structures. This allows structural properties expressed in temporal logics to be checked using standard model checking tools such as CWB [9]. The Kripke structures of the CWB are labelled transition systems generated from CCS process definitions. For this purpose, we use the Model Generator to convert applet graphs into a representation as CCS processes. Since CCS does not have the notion of valuation, atomic propositions p assigned to a node in an applet are represented by *probes*, that is, self–loops labelled by p. The translation also produces a set of process constants corresponding to the entry nodes of the respective applet. To model check an applet graph against a structural safety property, all initial states have to be checked individually. We encode the properties to be checked as μ–calculus formulae, replacing atomic propositions p by $\langle p \rangle\ \mathsf{true}$. Since CWB supports parametrised formulae, our specification patterns can directly be encoded.

When verifying $L \models_s \sigma_L$, we realised that in fact the choice of M_{lF} was too optimistic, as the implementation of *Loyalty.logFull* uses several other (internal) methods. Using the Applet Analyser we computed M_{lF} as the set of methods reachable from *Loyalty.logFull*, adapted the specification σ_L and reverified $L \models_s \sigma_L$. Reverifying the decomposition can be done automatically. The last column in Table 3 gives the verification times for model checking $P \models_s \sigma_P$ and $L \models_s \sigma_L$ on a Pentium 1.9 GHz machine.

7 Conclusions

This paper demonstrates a method to detect illicit interactions between applets, installed on a single smart card. The method is compositional, and therefore supports secure post–issuance loading of applets. In particular, the method allows to establish global control flow safety properties for a composed system, provided sufficient local properties are given for the applets. When the applets are loaded (post–issuance) it only remains to be shown that they respect their local property. while the global properties can be structural or behavioural, the local properties need to be structural. To support the specification process, a collection of specification patterns is proposed, with appropriate translations into the underlying logic.

We assembled a tool set – combining existing and newly developed tools – to support the verification tasks that arise in our method. Once the specifications are available, all verifications can be done using push–button technology. Thus, it can be automatically checked whether an applet can be accepted on the card.

The case study shows that the presented verification method and tool set can be used in practice for guaranteeing absence of illicit applet interactions. However, there are some possibilities for improvement. Finding suitable local properties, which requires ingenuity, is complicated by the requirement of formulating local properties structurally. Another difficulty stems from the inherent algorithmic complexity of two of the tasks: both maximal model construction and model checking behavioural properties are problems exponential in the size of the formula, thus making optimisations of these algorithms crucial for their successful application. For some common property patterns such as *Everywhere* σ, the size of the formula depends on the size of the interface. Therefore, it is crucial to develop abstraction techniques to abstract away from method names which are irrelevant to the given property.

Future work will thus go into fine–tuning the notion of interface, by defining public and private interfaces. Now interfaces contain all methods provided and required by a method. We wish to restrict the verification of the global safety properties to public interfaces, containing only the externally visible methods, provided and required by an applet. In order to check whether an implementation respects its local property, we will need to define an appropriate notion of hiding. We also intend to extend the set of specification patterns that we use, by investigating which classes of security properties generally are used. Finally, on a more theoretical side, we will study if we can extend the expressiveness of the logic used (*e.g.* by adding diamond modalities) and under what conditions we can allow behavioural local properties.

References

1. G. Barthe, D. Gurov, and M. Huisman. Compositional verification of secure applet interactions. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering 2002*, number 2306 in LNCS, pages 15–32. Springer, 2002.
2. F. Besson, T. Jensen, D. Le Métayer, and T. Thorn. Model checking security properties of control flow graphs. *J. of Computer Security*, 9(3):217–250, 2001.
3. A. Bouajjani, J.C. Fernandez, S. Graf, C. Rodriguez, and J. Sifakis. Safety for branching time semantics. In *Automata, Languages and Programming*, pages 76–92, 1991.
4. Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *International Conference on Concurrency Theory*, pages 135–150, 1997.
5. G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder - second generation of a Java model checker. In *Workshop on Advances in Verification*, 2000.
6. E. Bretagne, A. El Marouani, P. Girard, and J.-L. Lanet. Pacap purse and loyalty specification. Technical Report V 0.4, Gemplus, 2000.
7. O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 545–623. North Holland, 2000.
8. G. Chugunov, L.-å. Fredlund, and D. Gurov. Model checking of multi-applet JavaCard applications. In *CARDIS'02*, pages 87–95. USENIX Publications, 2002.
9. R. Cleaveland, J. Parrow, and B. Steffen. A semantics based verification tool for finite state systems. In *Proc. 9th IFIP Symp. Protocol Specification, Verification and Testing*, 1989.
10. J. Corbett, M. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *SPIN Model Checking and Software Verification*, number 1885 in LNCS. Springer, 2000.
11. J. Hatcliff and M. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. Technical report, SAnToS Laboratory, Department of Computing and Information Sciences, Kansas State University, 2000.
12. D. Kozen. Results on the propositional μ-calculus. *TCS*, 27:333–354, 1983.
13. D. Polanský. Verifying properties of infinite-state systems. Master's thesis, Masaryk University, Faculty of Informatics, Brno, 2000.
14. C. Sprenger, D. Gurov, and M. Huisman. Simulation logic, applets and compositional verification. Technical Report RR-4890, INRIA, 2003.
15. R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java Optimization Framework. In *CASCON 1999*, pages 125–135, 1999.

A Tool-Assisted Framework for Certified Bytecode Verification

Gilles Barthe and Guillaume Dufay

INRIA Sophia-Antipolis, France
{Gilles.Barthe,Guillaume.Dufay}@sophia.inria.fr

Abstract. Bytecode verification is a key security function in several architectures for mobile and embedded code, including Java, JavaCard, and .NET. Over the last few years, its formal correctness has been studied extensively by academia and industry, using general purpose theorem provers. Yet a recent roadmap on smartcard research [1], and a recent survey of the field of Java verification [11], point to a severe lack of methodologies, techniques and tools to help such formal endeavours. In earlier work, we have developed, and partly automated, a methodology to establish the correctness of static analyses similar to bytecode verification. The purpose of this paper is to complete the automation process by certifying the different dataflow analyses involved in bytecode verification, using the Coq proof assistant. It enables us to derive automatically, from a reference virtual machine that performs verification at run-time, and satisfies minimal requirements, a provably correct bytecode verifier.

1 Introduction

Several architectures for mobile and embedded code, including Java, JavaCard, and .NET, feature a bytecode verifier (BCV), which performs a modular (i.e. method per method) static analysis on compiled programs prior to their loading, and rejects potentially insecure programs that may violate type safety, or perform illegal memory accesses, or not respect the initialization protocol, or yield stack underflows or overflows, *etc.*

The bytecode verifier is a key security function in these architectures, and as such its design and implementation must be correct. Over the last few years, a number of projects have been successful in proving formally that bytecode verification is correct, in the sense that it obeys the specification of Sun. Many of these projects were carried by smartcard industrials in the context of security evaluations; for example, Schlumberger Cards and Terminals was recently awarded an EAL7 Common Criteria[1] certificate for their formal models of the JavaCard platform.

[1] The Common Criteria is an international evaluation scheme for the security of IT products. It features seven evaluation assurance levels (EAL). The highest quality levels EAL5 to EAL7 impose the use of formal methods for the modelling, specification and verification of the product being certified.

Yet such projects are labour-intensive, hence costly, and can only be conducted by a few experts in formal methods. A recent roadmap on smartcard research [1], and a recent survey of the field of Java verification [11], point to a severe lack of methodologies, techniques and tools to conduct cost-effective security evaluations.

Our work aims at developing tools and libraries that help validate execution platforms for mobile and embedded code. The research reported here focuses on specifying and proving the correctness of bytecode verification, following a methodology that is common to many existing works in the area, see e.g. [2, 4, 5, 16]. The methodology factors the effort into two phases:

– *virtual machines specification and cross-validation (VM phase):* during this first phase, one provides the specification of several virtual machines, including a defensive virtual machine that manipulates typed values and performs type-checking at run-time (as well as computations), and abstract virtual machine that only manipulates types and only performs type-checking. One also cross-validates these different virtual machines, e.g. one shows that the abstract virtual machine detects all typing errors that may occur during the execution of a program on the defensive virtual machine;
– *bytecode verification specification and verification (BV phase):* during this second phase, one builds and validates the bytecode verifier, using the abstract virtual machine defined during the VM phase. First, it involves modeling a dataflow analysis – for an unspecified execution function that meets minimal requirements. Second, it involves proving the correctness of the analysis; essentially it amounts to showing that the analysis will reject all programs that go wrong during execution. Third, it involves instantiating the analysis to the virtual machine defined in the VM phase, using cross-machine validation to establish that the VM enjoys all properties assumed for the unspecified execution function in the definition of the dataflow analyses; the instantiation provides a bytecode verifier, and a proof of its correctness. Note that different algorithms may be chosen, so as to account for some advanced features, e.g. subroutines and initialization in the Java and JavaCard platforms, or to minimize resource usage during verification, see Section 2.

In earlier work, we have been developing Jakarta, an environment which supports the specification and cross-validation of the virtual machines [3], and offers a high level of automation for performing the VM phase. In a nutshell, Jakarta consists of a specification language JSL in which virtual machines can be described, an abstraction engine that transforms virtual machines (e.g. that extracts an abstract virtual machine from a defensive one), and an interface with theorem provers, which maps JSL specifications to the prover specification language and generates automatically correctness proofs for the cross-validation of virtual machines. We mostly use Jakarta in conjunction with the proof assistant Coq [8], although prototypes interfaces to Isabelle [18] and PVS [20] exist[2].

[2] One particular reason for our choice is that French evaluation bodies recommend the use of Coq or B to carry Common Criteria evaluations at the highest levels.

The purpose of this paper is to complement our earlier work by providing a modular framework for performing the BV phase. Starting from an abstract notion of virtual machine on which we only impose minimal assumptions, we build a parametric bytecode verifier that encompasses a large class of algorithms for bytecode verification, and show the algorithm to be correct in the sense that it will reject programs that may go wrong. One novelty of our framework is to provide a high-level proof that it is sound to perform bytecode verification on a method per method basis. Another novelty is to provide a generic bytecode verifier that can be instantiated to several analysis including standard analyses that only accept programs with monomorphic subroutines, set-based analyses that accept programs with polymorphic subroutines, as well as other analyses for which no formal correctness proof was previously known. From a more global perspective, the combination of our framework for bytecode verification with the Jakarta toolset yields an automated procedure to derive a certified bytecode verifier from a reference defensive virtual machine; the procedure is applicable to many settings, and has been successfully used to certify the JavaCard platform. We return to these points in the conclusion.

Contents of the Paper. The remaining of the paper is organized as follows. We begin in Section 2 with a brief introduction to Coq and its module system, and with a brief overview of bytecode verification. We proceed in Section 3 with the basic definitions and constructions underlying bytecode verification. Section 4 and Section 5 are respectively devoted to formalizing and certifying a parameterized verification algorithm and compositional techniques that justify the method-per-method verification suggested by Sun. In Section 6, we show how the framework may be instantiated to different analyses. We conclude in Section 7 with related work, a general perspective on our results thus far, and directions for further research.

2 Preliminaries

2.1 Principles and Algorithms of Bytecode Verification

Bytecode verification [9, 17] is a static analysis that is performed method per method on compiled programs prior to their loading. Its aim is to reject programs that violate type safety, perform illegal memory accesses, do not respect the initialization protocol, yield stack underflows or overflows, *etc.*

The most common implementation of bytecode verification is through a dataflow analysis [13] instantiated to the abstract virtual machine that operates at the type level. The underlying algorithm relies on a history structure, storing the computed abstract states for each program point, and on an unification function on states. Then, starting from the initial state for a method, it computes a fixpoint with the abstract execution function. If the error state does not belong to the resulting history structure then bytecode verification is successful.

In the standard algorithm, called monovariant analysis, the history structure only stores one state for each program point and the unification function unifies, performing a join on the JavaCard type lattice, the computed state (resulting from one step of abstract execution) and the stored state. Unfortunately, this algorithm does not accept polymorphic subroutines (subroutines called from different program points). To handle such subroutines, the history structure must contain a set of states for each program point. For the polyvariant analysis, the unification function adds the computed state to the corresponding set from the history structure. This technique needs much more memory than monovariant analysis, however, it is possible to perform state unification rather than set addition in most cases. This last technique, called hybrid analysis (as described in [7, 12]), offers the best compromise between memory consumption, precision and efficiency.

Our framework also deals with lightweight bytecode verification [19], a special kind of verification that can fit and run in chips used for smart cards, but due to space constraints details are omitted.

2.2 The Coq Proof Assistant

Coq [8] is a general purpose proof assistant which is based on the Calculus of Inductive Constructions, and which features a very rich specification language and a higher-order predicate logic. However, we only use neutral fragments of the specification language and the logic, i.e. fragments which are common to several proof assistants, including Isabelle and PVS. More precisely, we use first-order logic with equality, first-order data types, structural recursive definitions, record types, but no dependent types – except in the definition of `gfp`, but such a function is also definable in Isabelle and PVS. Furthermore, Coq underlying logic is intuitionistic, hence types need not have a decidable equality. For the sake of readability, and because it is specific to Coq, we gloss over this issue in our presentation[3].

Modules. Our framework makes an extensive use of the interactive ML-style modules that were recently integrated to Coq [6]. Hence we briefly review the syntax for modules. The keyword **`Module Type`** introduces the declaration of a type of a module, and is followed by its name, a collection a **`Parameter`** and **`Axiom`** declarations giving its signature, and it is closed by the keyword **`End`**. A module type can also include (and, in a certain sense, extend) other module types with the keyword **`Declare Module`**. A module type is implemented using the keyword **`Module`** (the module type it satisfies is specified after the notation `<:`). As usual, the module must fulfill the signature of the module type it implements. Note that other modules can be given as parameters of a module. Finally, constructions of a module can be accessed outside the module using the dot notation of qualified names or directly with the keyword **`Import`** followed by the module name.

[3] Although our framework addresses decidability by making appropriate assumptions in modules, we omit such assumptions in this paper.

Notations. The type of propositions is `Prop`, and the type of data is `Set`. The types `predicate A` and `relation A` respectively denote the set of predicates and binary relations over a type `A`.

We conclude with some basic definitions used throughout the paper. Given `A : Set`, $<_A$`: (relation A)`, `f : A`$\rightarrow$`A` and `P : (predicate A)`, we let $\leq_A$ denote the reflexive closure of $<_A$ and define

$$\begin{aligned}
\texttt{(monotone } <_A \texttt{ f)} &\equiv \forall \texttt{a,a':A.(a } <_A \texttt{ a')} \rightarrow \texttt{((f a)} \leq_A \texttt{ (f a'))}\\
\texttt{(decreases } <_A \texttt{ f)} &\equiv \forall \texttt{a:A.((f a)} \leq_A \texttt{ a)}\\
\texttt{(down\_closed } <_A \texttt{ P)} &\equiv \forall \texttt{a,a':A.(a } <_A \texttt{ a')} \rightarrow \texttt{(P a')} \rightarrow \texttt{(P a)}
\end{aligned}$$

Finally we let `(well_founded` $<_A$`)` state that the relation $<_A$ is well founded, i.e. that there is no infinite decreasing chain.

3 Bytecode Verification as a Fixpoint Computation

We favour a definition that abstracts away from implementation details, and define a bytecode verifier as a predicate that rejects programs that may go wrong.

Definition 1.

– *A* transition system with error *(TSE) is given by a type* `state` *of* states, *an* execution relation `exec` *over states, and a set* `err` *of error states. Formally,*

```
Module Type TSE.
Parameter state : Set.
Parameter exec  : (relation state).
Parameter err   : (predicate state).
End TSE.
```

We say that a state `a` *of a given TSE is bad, written* `bad a`*, if it can reach an error state by successive transitions of the execution relation.*

– *A* bytecode verifier *over a module* `tse` *of type* `TSE` *is given by a predicate* `check` *that rejects all bad states. Formally, the module* `BCV` *of bytecode verifiers extends the module* `TSE` *as follows*[4]*:*

```
Module Type BCV.
Declare Module tse: TSE. Import tse.
Parameter check : (predicate state).
Axiom ∀ a:state.(check a) →¬(bad a).
End BCV.
```

The standard way to build a bytecode verifier is to endorse the type of states with a well-founded order for which execution is decreasing (to guarantee termination), and such that error states are downwards closed. If furthermore execution is deterministic, one can compute for every state `a`, the greatest fixpoint `b` below `a`; then it is sufficient to check that `b` is not an error state to conclude that `a` is not bad.

[4] Coq modules provide names for axioms, so that these axioms can later be used in proofs. For readability we omit names of axioms in our module declarations.

Definition 2. *A* fixpoint structure with errors *(FSE) is given by the module*

```
Module Type FSE.
Parameter state   : Set.
Parameter exec    : state → state.
Parameter err     : (predicate state).
Parameter <state  : (relation state).

Axiom (well_founded <state).
Axiom (decreases <state exec).
Axiom (monotone <state exec).
Axiom (down_closed <state err).
End FSE.
```

We can define a module functor satisfying, from a module of type FSE, the type of the module BCV:

```
Module FSE2BCV [fse:FSE] <: BCV.
```

To do so, we first define for every state `a` of a FSE the greatest fixpoint `gfp a` below it as

$$\texttt{gfp a} = \begin{cases} \texttt{a} & \text{if } \texttt{exec a = a} \\ \texttt{gfp (exec a)} & \text{otherwise} \end{cases}$$

Then, we define `check a` as $\neg$`(err_state (gfp a))`. As execution is monotone and `gfp a` is the greatest fixpoint below `a`, it is clear that such a checking is sufficient to guarantee that `a` is not a bad state.

4 A Parameterized Bytecode Verifier

In this section, we construct a parameterized bytecode verifier that rejects programs that may go wrong when executed with an abstract virtual machine. We start with the definition of the latter.

Definition 3. *An abstract virtual machine (AVM) is given by an ordered type of states* `state` *endorsed with a downwards closed set of errors* `err`*, a type of locations* `loc`*, an execution function* `exec`*, a successor function* `succs` *that computes the successors of a state and an enumeration* `locs` *of the locations of the program. Formally,*

```
Module Type AVM.
Parameter state : Set.
Parameter <state : (relation state).
Parameter err   : (predicate state).
Parameter loc   : Set.
Parameter succs : loc → state → (list loc).
Parameter locs  : (list loc).
Parameter exec  : loc → state → state.

Axiom (down_closed <state err).
End AVM.
```

Bytecode verification relies on stackmaps, i.e. functions that associate to every program point a history structure. History structures can be seen as an abstraction of the mathematical set notion.

Definition 4. *The module type* `History_Struct` *of* history structures *is parameterized by a carrier set* `A`[5] *and given a type constructor* `hist`, *a projector function* `hist_unit`, *an extension* `hist_less` *of an order on* `A`, *an decreasing iterator* `hist_foldr` *and a membership predicate* $\in_{\texttt{hist}}$ *on which we define an existential predicate* $\exists_{\texttt{hist}}$. *Formally,*

```
Module Type History_Struct [A:Set].
Parameter hist      : Set → Set.
Parameter hist_unit : A → (hist A).
Parameter hist_less : ∀ <_A:(relation A). (relation (hist A)).
Parameter ∈_hist     : A →(hist A) →Prop.
Axiom ∀ x:A.(x ∈_hist (hist_unit x)).

Parameter hist_foldr : ∀ B:Set.(A→B→B) → B→(hist A)→ B.
Axiom ∀ B:Set. ∀ f:(A →B →B). ∀ <_B:(relation B).
 (∀ a:A.(decreases <_B (f a))) →
 (∀ a:(hist A).(decreases <_B (λ b:B.(hist_foldr B f b a)))).

Definition ∃_hist := λ P:(predicate A) λ s:(hist A)
  ∃ a.(P a) ∧ (a ∈_hist s).
Axiom ∀ <_A:(relation A). ∀ P:(predicate A).
  (down_closed <_A P) →(down_closed (hist_less <_A) (∃_hist P)).
End History_Struct.
```

In the following, we will use the notation $<_{\texttt{A}}^{\texttt{hist}}$ for the extension `(hist_less <_A)` of a relation $<_{\texttt{A}}$.

The definition of stackmaps is included in a module of stackmap structures. Essentially, a stackmap structure consists of an abstract virtual machine, of a history structure, and of a unification function that merges states and that is decreasing and monotone w.r.t. the order inherited from the history structure. In order to construct a fixpoint structure from a stackmap structure, we also require histories $<_{\texttt{A}}^{\texttt{hist}}$ to be well-founded and a supremum for $<_{\texttt{A}}$.

Definition 5. *The module SMS of* stackmap structures *is defined as:*

```
Module Type SMS.
Declare Module avm : Abstract_VM. Import avm.
Declare Module hs  : (History_Struct state). Import hs.

Parameter unify : state → (hist state) → (hist state).
Axiom ∀ s:state.(decreases <^hist_state (unify s)).
Axiom ∀ s:state.(monotone <^hist_state (unify s)).
Axiom ∀ s:state.∀ s':(hist state).(unify s s')=s' →
∃ y.(y ≤_state s) ∧ (y ∈_hist s').
```

[5] In reality, Coq modules type can only be parameterized by other modules, so one has to use a module that is "isomorphic" to `Set`.

```
Parameter ⊤ : state.
Axiom ∀ a:state.(a ≤state ⊤).

Axiom (well_founded <hist state).
End SMS.
```

One can define the type stackmap of stackmaps over a stackmap structure sms as list (sms.avm.loc * (sms.hs.hist sms.avm.state)), and an execution function over stackmaps sms_exec: stackmap →stackmap which corresponds to the recursive procedure in Kildall's algorithm [13]. It is straightforward to derive a fixpoint structure, and hence a bytecode verifier, for the TSE induced by sms_exec: stackmap →stackmap. In order to conclude that the resulting fixpoint structure also yields a bytecode verifier for the TSE induced by the AVM, one needs to observe that the following diagram commutes:

$$\begin{tikzcd}
\texttt{avm.loc*avm.state} \arrow[r, "\texttt{avm.exec*avm.next}"] \arrow[d, "\texttt{make\_stackmap}"] & \texttt{avm.loc*avm.state} \arrow[d, "\leq\texttt{make\_stackmap}"] \\
\texttt{stackmap} \arrow[r, "\texttt{sms\_exec}"] & \texttt{stackmap}
\end{tikzcd}$$

Here make_stackmap denotes the function that takes as input a pair $\langle \texttt{l}, \texttt{a} \rangle$, and returns as output the stackmap in which the program point l is associated to the singleton history hist_unit a, and every other program point is associated to ⊤.

5 Correctness of Bytecode Verification

In the previous section, we have shown that programs that pass bytecode verification do not go wrong when executed on an abstract virtual machine which satisfies minimal requirements. The purpose of this section is to lift this result to a defensive virtual machine: more precisely, we are going to show that programs that pass bytecode verification do not go wrong when executed on a defensive virtual machine which satisfies minimal requirements. It involves relating a defensive and an abstract virtual machine, and proving that no difficulty arises through exception handling (which is performed by the defensive virtual machine, but ignored by the abstract one), or through method invokation (which remains within the same frame for the abstract virtual machine, as explained below). We begin by defining defensive virtual machines.

Definition 6. *A* defensive virtual machine *(DVM) is given by a type* state *of states, an execution function* exec*, a type* frame *of frames, an accessor function* getstack *that associates to each state a list of frames (i.e. its stack), another accessor function* getinstr *that associates to each state the nature of the next execution to be executed, and a set* err_frame *of error frames. Formally,*

```
Module Type DVM.
Parameter state     : Set.
Parameter exec      : state → state.
Parameter frame     : Set.
Parameter getstack  : state → (list frame).
Parameter getinstr  : state → type_of_instr.
Parameter err_frame : (predicate frame).
End DVM.
```

The function `getinstr` distinguishes between 4 cases: execution is intra-procedural `sameframe` (that acts only in the current frame, e.g. for arithmetic instruction, branching instruction, *etc*), execution is a method invokation `invoke`; execution is a return step `return` (pops a frame); or execution raises an exception `exception`.

We now turn to formulating a set of general properties about method invokation and exception handling, and proving that such properties ensure that programs that pass bytecode verification will not go wrong. These properties involve an abstract virtual machine and an abstraction function.

Abstract Virtual Machine. We assume given an abstract virtual machine `avm`, with a function `init` that returns for each method or exception the corresponding initial state. Furthermore, we assume given a decomposition of abstract method invokation in two functions, so as to be able to simulate the modifications made by the concrete virtual machine on a frame when the control flow is given to the invoked method and when it returns to the invoker method. Formally, we assume given two functions `exec_invk` and `exec_ret` whose composition is equal to `avm.exec` for states `a` such that `getinstr a = invoke`.

Abstraction Function. We assume given a function that maps a frame to an abstract state and a location α: `dvm.frame` →`avm.state`. The function is extended a function β: `dvm.state` →`avm.state`. on defensive states by abstracting the topmost frame of the stack (if the stack is empty, we return a default error value).

Safe States. We now turn to the definition of safe abstract states. A abstract state will be safe if it is greater than a state belonging to the history structure computed by the abstract bytecode verifier at the location of the given state. This notion is extended to defensive frames by abstraction.

The notion of safety for a defensive state must guarantee that the stack is well-formed, i.e. that all the frames below the top one are in an "intermediate" state which is not reached by the abstract virtual machine until the invoked method returns. Then, a defensive state `s` will be safe if lstinline!getstack s = []
! or if `getstack a = f::lf` and each frame in `lf` is of the form `exec_invk f'` where `f'` is a safe frame.

We now show that safe states are closed under execution and are not bad.

Lemma 1. *Let* `s` *be a defensive state. Suppose:*

- *if* `getinstr s = sameframe`, *then the following property holds:*

$$\texttt{(avm.exec (}\beta\texttt{ s))} \leq_{\texttt{state}} \texttt{(}\beta\texttt{ (dvm.exec s))}$$

- *if* `getinstr s = invoke` *and* `getstack s = f::lf`, *then the following properties hold:*

$$\exists\, \texttt{f':frame. getstack (dvm.exec f') = f'::(exec\_add f)::lf}$$
$$\texttt{(init (}\beta\texttt{ s))} \leq_{\texttt{state}} \texttt{(}\beta\texttt{ s)}$$

- *if* `getinstr s = return` *and* `getstack (dvm.exec s) = f::lf`, *then there exists two frames* `f'` *and* `f''` *such that the following properties hold:*

$$\texttt{getstack s = f'::f''::lf}$$
$$\texttt{(aexec\_ret (}\alpha\texttt{ f''))} \leq_{\texttt{state}} \texttt{(}\alpha\texttt{ f)}$$

- *if* `getinstr s = exception` *and* `getstack s = f::(lf@lf')`, *then the following properties hold:*

$$\exists\, \texttt{f':frame. getstack s = f'::lf'}$$
$$\texttt{(init (}\beta\texttt{ s))} \leq_{\texttt{state}} \texttt{(}\beta\texttt{ s)}$$

If furthermore `s` *is safe, then* `dvm.exec s` *is also safe.*

This lemma if proved using properties on the bytecode verifier and property on `exec_invk` and `exec_ret` w.r.t. `avm.exec`. Then one can easily construct a bytecode verifier for a defensive virtual machine `dvm`. Formally, from a module type `Comp_Struct`, that contains all the assumptions of Lemma 1, we are able to define a functor module `BCV_dexec` satisfying the module type `BCV` for the execution `dvm.exec`. The function `check` of the module type `BCV` is defined assuming that the given defensive state is safe and that the result of bytecode verification for all initial states (methods and exceptions) of the program does not contain an error state. Finally, by Lemma 1, we can prove the property `check_ok` of the module type `BCV`, stating that if the verification `check` was successful, we can not reach with the defensive virtual machine an error state.

6 Instantiation of History Structures

The previous section describes the construction of a correct bytecode verifier for a defensive virtual machine. The construction is parameterized by a structure that records the history of the computations performed by the verifier. The purpose of this section is to present different instantiations of our framework, focusing on different choices of history structures that correspond to the algorithms described in Section 2. We use these instantiations as convenient entry points in our formalization, see Subsection 7.2.

Monovariant Analysis. A *monovariant analysis* (MA) is given by a well-founded order on states with a supremum, and by proofs that the execution function is monotone w.r.t. the order on states and the unification is decreasing and monotone. Formally,

```
Module Type Monovariant_Analysis_Struct.
Declare Module avm : Abstract_VM. Import avm.

Parameter unify  : state → state → state.

Axiom ∀ s:state.(decreases <_state (unify s)).
Axiom ∀ s:state.(monotone <_state (unify s)).
Axiom ∀ s,s':state.(unify s s')=s' →∃ y.(y ≤_state s) ∧ (y ∈_hist s').

Parameter ⊤ : state.
Axiom ∀ a:state.(a ≤_state ⊤).

Axiom (well_founded <_state).
Axiom ∀ l:loc.(monotone <_state (exec l)).
End Monovariant_Analysis_Struct.
```

The construction of a stackmap structure from a monovariant analysis is done by a functor module `Monovariant_Analysis` from the previous module type definition. It mainly proceeds by instantiating the parametric history structure to the identity history structure, in which `hist A` is defined as `A`, and the other fields are instantiated in the obvious way.

Polyvariant Analysis. A *polyvariant analysis* is given by a natural number `max_length_set` that fixes the maximal size of the set of abstract states associated to each program point, by a supremum state ⊤, by an error state `err_st` and by a proof that execution is monotone. Formally,

```
Module Type Polyvariant_Analysis_Struct.
Declare Module avm : Abstract_VM. Import avm.

Parameter max_length_set : nat.
Parameter err_st : state.
Axiom (err err_st).
Parameter ⊤ : state.
Axiom ∀ a:state.(a ≤_state ⊤).

Axiom ∀ l:loc.(monotone <_state (exec l)).
End Polyvariant_Analysis_Struct.
```

One proceeds by instantiating the history structure in such a way that `hist A` is defined as the set of elements of `A` of cardinal less than `max_length_set` (the other fields are instantiated in the obvious way). Then, this module is used with the functor `Polyvariant_Analysis` to construct a stackmap structure, defining the function `unify` as:

```
λ a:state λ s:(hist state)
(if ((set_size (set_add a s)) < max_length_set)
 then (set_add a s)
 else (set_add err_st s))
```

In that case, `hist_less` does not use the order $<_{\mathtt{state}}$ and is defined as set inclusion. It is interesting to notice that the polyvariant analysis is by far the simplest algorithm to instantiate.

Hybrid Analysis. An *hybrid analysis* is given combining elements needed by monovariant and hybrid analysis and adding an optimization function `opt_unify` to discriminate in which cases the unification of states must take place. Formally,

```
Module Type Hybrid_Analysis_Struct.
Declare Module avm : Abstract_VM. Import avm.

Parameter opt_unify : state → state → bool.
Parameter unify  : state → state → state.

Axiom ∀ s:state.(decreases <_state (unify s)).
Axiom ∀ s:state.(monotone <_state (unify s)).
Axiom ∀ s,s':state.(unify s s')=s' →∃ y.(y ≤_state s) ∧ (y ∈_hist s').

Parameter max_length_set : nat.
Parameter err_st : state.
Axiom (err err_st).
Parameter ⊤ : state.
Axiom ∀ a:state.(a ≤_state ⊤).

Axiom (well_founded <_state).
Axiom ∀ l:loc.(monotone <_state (exec l)).
End Hybrid_Analysis_Struct.
```

The same history structure as polyvariant analysis is used for the hybrid analysis. The function `unify` is then defined as follow :

```
λ a:state λ s:(hist state)
(Case (set_map_bool opt_unify unify a s) of
  (Some res) ⇒ res |
   None ⇒ (if ((set_size (set_add a s)) < max_length_set)
             then (set_add a s)
             else (set_add err_st s))
 end)
```

where `set_map_bool` ranges over elements of `s`, performs unification depending on the result of `opt_unify` and returns the resulting set if unification has occurred or `None` otherwise. In that case, `hist_less` combines the order $<_{\mathtt{state}}$ on states and set inclusion.

7 Conclusions

7.1 Related Work

As mentioned in the introduction, there is a considerable body of machine-checked specifications of execution platforms such as the JVM or .NET, many of which use the methodology instrumented in our work. A notable exception is the extensive account of bytecode verification developed by Klein, Nipkow, and Wildermoser [14, 15] using the proof assistant Isabelle [18]. For lack of space, we refer the reader to [11, 17] for a more comprehensive account of related work.

There are also machine-checked proofs of type soundness for .NET [10, 21]. This work is more closely related to ours in the sense that [21] explicitly aims at developing tools to automate type soundness proofs. The major difference with our work is that they do not pursue cross-machine validation, and opt instead for a standard type soundness proof.

7.2 Perspectives

We have develop a general framework that establishes the correctness of a parameterized bytecode verifier, and justifies the compositional techniques of bytecode verification. The framework has been instantiated for specific history structures that are often considered in the literature and implementations. These instantiations provide convenient entry points to our framework, and can be used in combination with Jakarta to build and validate bytecode verifiers with a high degree of automation. As illustrated in Figure 1, such a combination requires the user to provide:

– a defensive virtual machine;
– the definition of abstraction functions, in the form of Jakarta abstraction scripts, that are used to construct the abstract virtual machine and an offensive virtual machine[6]. Scripts may contain some minimal amount of proof information to carry cross-machine validation;
– a formal proof of the correctness w.r.t. bytecode verification of method invokation and exception handling, i.e. an instantiation of the module `Struct_Comp` of Section 5;
– an instantiation of the history modules to the abstract virtual machine generated by Jakarta;

and returns an offensive virtual machine, several bytecode verifiers, and a proof that these bytecode verifiers are correct, in the sense that they will reject programs that go wrong on the defensive virtual machine, and that the offensive and defensive virtual machines coincide on programs that are accepted by bytecode verification.

Such a combination has been used to good purpose for validating the JavaCard platform. Using Jakarta, we have generated from a defensive virtual machine

[6] Such a machine manipulates untyped values, and relies on the bytecode verifier to detect programs that may go wrong.

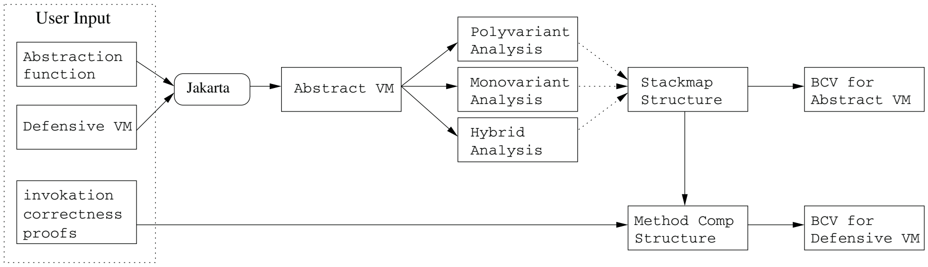


Fig. 1. Framework architecture

(10,000 lines of code), both an abstract and an offensive virtual machine (5,000 lines of code each), as well as more than 10,000 lines of proof scripts that establish cross-machine validation and the monotonicity of the generated abstract virtual machine. We have provided another 1,500 lines of proof scripts which concern the correctness w.r.t. bytecode verification of method invokation and exception handling. Together with the output of Jakarta, these 1,500 lines provide all relevant information for the bytecode verifier to be proved correct – without any further user interaction.

As to future work, we plan to instantiate our framework to enhanced bytecode verifiers that guarantee a stronger security of applications. Indeed, there have been many proposals of type systems for the JVM that provide stronger guarantees with respect to safety and security, and it would be interesting to adapt our virtual machine specifications to such type systems, and use the framework described here to derive certified bytecode verifiers based on these type systems. In fact, we have started modeling a defensive JVM machine for an information flow type system[7], and intend to use the framework described in this paper, in combination with Jakarta, to build and validate a bytecode verifier for information flow. Likewise, it would be interesting to apply our methodology to other execution platforms, such as the .NET platform.

References

1. Roadmap for European Research on Smartcard Technologies. See `http://www.ercim.org/reset`
2. J. Andronick, B. Chetali, and O. Ly. Using Coq to Verify Java Card Applet Isolation Properties. In D. Basin and B. Wolff, editors, *Proceedings of TPHOLs'03*, volume 2758 of *Lecture Notes in Computer Science*, pages 335 – 351. Springer-Verlag, 2003.

[7] Non-interference is a property about all program executions and thus it cannot be completely enforced by a defensive virtual machine. However, we only use the defensive virtual machine as a tool to prove that the bytecode verifier enforces non-interference, without making any claim on the security guarantees provided by such a defensive virtual machine.

3. G. Barthe, P. Courtieu, G. Dufay, and S. Melo de Sousa. Tool-Assisted Specification and Verification of the JavaCard Platform. In H. Kirchner and C. Ringessein, editors, *Proceedings of AMAST'02*, volume 2422 of *Lecture Notes in Computer Science*, pages 41–59. Springer-Verlag, 2002.
4. G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. Melo de Sousa. A Formal Executable Semantics of the JavaCard Platform. In D. Sands, editor, *Proceedings of ESOP'01*, volume 2028 of *Lecture Notes in Computer Science*, pages 302–319. Springer-Verlag, 2001.
5. G. Betarte, B. Chetali, E. Giménez, C. Loiseaux, and O. Ly. Formal Modeling and Verification of the Java Card Security Architecture: from Static Checkings to Embedded Applet Execution. In *Proceedings of ESMART'02*, 2002.
6. J. Chrzaszcz. Implementing Modules in the Coq System. In D. Basin and B. Wolff, editors, *Proceedings of TPHOLs'03*, volume 2758 of *Lecture Notes in Computer Science*, pages 270 – 286. Springer-Verlag, 2003.
7. A. Coglio. Simple Verification Technique for Complex Java Bytecode Subroutines. In *Proceedings of FTFJP'02*, 2002.
8. Coq Development Team. *The Coq Proof Assistant User's Guide. Version 7.4*, February 2003.
9. S. N. Freund and J. C. Mitchell. A Type System for the Java Bytecode Language and Verifier. *Journal of Automated Reasoning*, 30(3-4):271–321, December 2003.
10. A.D. Gordon and D. Syme. Typing a multi-language intermediate code. In *Proceedings of POPL'01*, pages 248–260. ACM Press, 2001.
11. P. Hartel and L. Moreau. Formalizing the Safety of Java, the Java Virtual Machine and Java Card. *ACM Computing Surveys*, 33(4):517–558, December 2001.
12. L. Henrio and B. Serpette. A parameterized polyvariant bytecode verifier. In J.-C. Filliatre, editor, *Proceedings of JFLA'03*, 2003.
13. G. A. Kildall. A unified approach to global program optimization. In *Proceedings of POPL'73*, pages 194–206. ACM Press, 1973.
14. G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, April 2002.
15. G. Klein and M. Wildmoser. Verified bytecode subroutines. *Journal of Automated Reasoning*, 30(3-4):363–398, December 2003.
16. J.-L. Lanet and A. Requet. Formal Proof of Smart Card Applets Correctness. In J.-J. Quisquater and B. Schneier, editors, *Proceedings of CARDIS'98*, volume 1820 of *Lecture Notes in Computer Science*, pages 85–97. Springer-Verlag, 1998.
17. X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, December 2003.
18. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
19. E. Rose and K. H. Rose. Lightweight bytecode verification. In *Workshop "Formal Underpinnings of the Java Paradigm", OOPSLA'98*, October 1998.
20. N. Shankar, S. Owre, and J.M. Rushby. *The PVS Proof Checker: A Reference Manual*. Computer Science Laboratory, SRI International, February 1993. Supplemented with the PVS2 Quick Reference Manual, 1997.
21. D. Syme and A. D. Gordon. Automating type soundness proofs via decision procedures and guided reductions. In M. Baaz and A. Voronkov, editors, *Proceedings of LPAR'02*, volume 2514 of *Lecture Notes in Computer Science*, pages 418–434, 2002.

Reasoning about Card Tears and Transactions in Java Card

Engelbert Hubbers and Erik Poll

SoS Group, NIII, Faculty of Science, University of Nijmegen
{hubbers,erikpoll}@cs.kun.nl

Abstract. The Java dialect Java Card for programming smartcards contains some features which do not exist in Java. Java Card distinguishes *persistent* and *transient* data (data stored in EEPROM and RAM, respectively). Because power to a smartcard can suddenly be interrupted by a so-called *card tear*, by someone removing the smartcard from the reader, Java Card provides a notion of *transaction* to ensure that updates of multiple fields in persistent memory can be performed atomically. This paper describes a way to reason about these Java Card specific language features.

1 Introduction

The Java Card language for programming smartcards has attracted a lot of attention in the formal methods community, especially people working on formal methods for Java. In many respects, it provides an ideal target for formal methods: the language and its API are simple, programs are very small, and their correctness is critical.

However, although the Java Card programming language for smartcards is usually presented as a subset of Java, Java Card has several features in addition to standard Java, which are specific to smartcards. First, Java Card distinguishes the two kinds of memory that are available on smartcards, persistent (EEPROM) and transient (RAM). Second, because a smartcard can be subject to a sudden loss of power due to a so-called *card tear* – namely when the card is removed from the reader – Java Card offers a *transaction mechanism* similar to that found in databases; this enables a programmer to ensure that several updates to memory are performed atomically, i.e. either all the updates are performed or none is. There are more Java Card specific features, but we do not take them into account in this paper.

To accurately reason about the behavior of Java Card programs – and to do program verification – these additional features should be taken into account. Most work on the verification of Java Card programs, with the exception of [1], ignores these special features. Given the complexities of program verification, this can certainly be justified for pragmatic reasons: before we try to verify that a program is correct in the presence of potential card tears, it makes sense to first verify its correctness under the simplifying assumption that no card tears occur and no transactions are ever aborted. However, ultimately we would like to be able to reason about such Java Card features, and this is what we set out to do in this paper.

The context of this work is the verification of Java programs that have been specified with JML [2], using the LOOP tool in combination with the theorem prover PVS. The verification of programs with the LOOP tool ultimately relies on a denotational semantics of Java and JML, for which a Hoare logic and weakest precondition calculus have been developed. In short, the LOOP tool compiles JML annotated Java source code into PVS theories. Proving these theories in PVS implies that it is formally verified that the Java program behaves the way it is specified in JML. For a more detailed overview of this LOOP project, see [3]. One of the achievements of this work has been that a commercial Java Card application has been completely verified, showing that such verifications of real Java Card programs are feasible. Still, our verifications ignore the possibility of card tears, so our next challenge is to take this into account.

To reason about card tears and transactions we need a formal semantics of these features (or a programming logic which takes them into account). Rather than defining a semantics of Java Card including these features from scratch, we will try to desugar Java Card programs with their special features into conventional Java programs, effectively modeling card tears and transactions inside Java. The central trick we use here is that we model card tears as special exceptions, a trick also used in [4,5]. Such a modular approach has several benefits (provided it is successful of course...): it is less work, it is easier to understand, and because it is independent of a particular semantics or programming logic for Java, it will be applicable in many other settings, not just the particular semantics and programming logic of Java that is used in the LOOP project. Modularity is not just a desirable quality for programs, but also for theories about programming languages!

The organization of the rest of this paper is as follows: Sect. 2 explains the peculiarities of Java Card that we want to reason about. Sect. 3, 4 and 5 describe our approach in detail. Sect. 6 says something about the implementation of our idea.

2 Card Tears and Transactions in Java Card

In this section we briefly explain the peculiarities of Java Card as opposed to Java when it comes to card tears and transactions. For a more complete explanation, see [6] or the Java Card Runtime Environment (JCRE) specification [7].

Persistent vs. Transient Memory. Java Card distinguishes two kinds of memory that are available on smartcards, persistent (EEPROM) and transient (RAM)[1]. The main difference is that persistent memory will keep its value when

[1] Smartcards will also have ROM, which is used for pre-installed program code, but this is of no concern to the Java Card programmer.

power is switched off. Java Card objects and their fields are allocated in EEPROM, so the fields of objects will keep their value during a power loss. However, the Java Card API offers methods to allocate arrays in RAM, so-called *transient arrays*. If a field is a transient array, then the contents of this array are lost as soon as power is lost, but the field itself, which is a reference to the piece of RAM allocated for the array, keeps its value as this is stored in EEPROM. Reasons for using RAM rather than EEPROM for (array) field are efficiency – reading and writing RAM is quicker than EEPROM –, the limited lifetime of EEPROM – EEPROM can only support a limited number of writes before the chips stops functioning –, and security – data kept in RAM is harder to spy out and moreover it is lost as soon as power is lost[2]. The stack is also stored in RAM, so the parameters and result of method calls and local variables are all lost as soon as power is lost.

Card Tears. In many card readers it is possible to tear the smartcard out of the reader while it is in operation. Such a so-called *card tear* results in a sudden loss of power. All data stored in RAM is lost when such a card tear occurs. The Java Card platform incorporates a special clean-up when power supply is restored, before any normal action applet operation takes place.

Transactions. To cope with card tears, the Java Card API offers a so-called *transaction mechanism*. This can be used to ensure that several updates to persistent memory are executed as a single atomic operation, i.e. either all updates are performed or none at all. The Java Card API offers three methods for this: `beginTransaction`, `commitTransaction` and `abortTransaction`. After a `beginTransaction` all changes to persistent data are executed conditionally. Note that changes to transient data, including local variables, are executed unconditionally. The transaction is ended by `commitTransaction` or `abortTransaction`; in the former case the updates are committed, in the latter case the updates are discarded. If a card tear occurs during a transaction, any updates to persistent data done during that transaction are discarded. This in fact happens the next time the smartcard powers up during the special clean-up mentioned before.

Example 1 (Java Card sample). Fig. 1 illustrates the use of the transaction mechanism, the use of the API method for allocating a transient array, and the use of JML to specify invariants and postconditions.

Every object of class `A` has a persistent field `p` and a field `t` that is a transient array of length 1. This means that whenever the smartcard loses power, the contents of `t[0]` is lost, but `p` and `t` itself –i.e. the pointer to the position in the RAM memory where `t[0]` is stored– keep their value.

[2] Indeed, for security reasons, the contents of transient arrays can also be cleared automatically at certain events other than card tears, e.g. the de-selection of an applet.

```
class A {
  // persistent field p, allocated in EEPROM
  byte p;
  //@ invariant p % 2 == 0 && 0<= p && p < 10;

  // transient array t, so t[0] is allocated in RAM
  byte[] t = makeTransientByteArray(1,CLEAR_ON_RESET);
  //@ invariant t != null && t.length == 1;
  //@ invariant t[0] % 2 == 0;

  //@ ensures p == \old(p)+2 && t[0] == \old(t[0])+2;
  void m() {
    beginTransaction();
    p++; t[0]++; p++;
    if (p < 10) commitTransaction();
           else abortTransaction();
    t[0]++;
    }
}
```

Fig. 1. Example program using transactions and transient data, with JML specification

There are four JML annotations in the example, written as comments starting with `//@`. This includes three invariants stating that `p` is even, that `t` is not null and has always length 1 and that `t[0]` is also even. This also includes one postcondition (`ensures` clause) for method `m`, stating that the method will increase the values of `p` and `t[0]` by 2. The use of the transaction guarantees that the invariant for `p` will not be broken if the method `m` is interrupted by a card tear. The treatment of transient memory makes sure that the invariants for `t` are not broken by a card tear. Note that the postcondition only relates to normal termination of the method, and does not say anything about what happens if the method 'aborts' because of a card tear.

Incorrect use of this mechanism can result in a `TransactionException` being thrown:

– The transactions cannot be nested. So if a new `beginTransaction` is called within another transaction a `TransactionException` is thrown. Likewise such an exception is thrown if a `commitTransaction` or `abortTransaction` is called while there is no transaction in progress.
  Reasoning about this requires no special machinery, as specifications for enforcing the correct use of the methods for beginning or ending transactions can easily be expressed in JML.
– A `TransactionException` is also thrown if certain hardware limitations are exceeded. Only a finite amount of storage, called the *commit buffer*, is available to keep track of the conditional updates done during a transaction. The size of this commit buffer depends on the specific smartcard hardware. If there are too many updates inside a transaction, and the available space in the commit buffer is exhausted, again a `TransactionException` is thrown. We will ignore the possibility of exhausting the commit buffer, and the resulting `TransactionException`. Proving that this never happens is best done in

an ad-hoc manner, i.e. by counting the maximum number of bytes needed in the commit buffer for every transaction in a program and checking that this does not exceed the space of the commit buffer. Including this in a general program logic would be overly complicated. Also, how much space needed in the commit buffer for the bookkeeping associated with an individual update will be specific to the particular implementation of the platform and/or the underlying hardware.

Some (native) classes in the Java Card API provide persistent data which is not subjected to card tears: the counter associated with a `PIN` object, which keeps track of how many incorrect `PIN`s have been entered, is not restored in the event of a card tear. Otherwise the transaction mechanism might allow an unlimited number of guesses for the `PIN` code.

2.1 What Can Go Wrong, and How to Avoid It

Before we consider ways of describing the semantics of card tears and transactions, and how this might be used as a basis for reasoning about these language features, we first approach the issue from a different angle, by investigating what can go wrong if code is subjected to card tears or if it contains transactions, and what could we do to avoid these problems. Or, in other words, what are the properties that we fail to establish in our current verifications of Java Card code, but which we would like to be able to prove.

Invariants. Invariants usually play a crucial role in ensuring that a piece of code behaves correctly. When a card tear occurs, invariants may be left broken as a result. After all, invariants may temporarily be broken during the execution of a method.
Typically, the transaction mechanism is used to prevent card tears from disturbing invariants that involve persistent data, as in Fig. 1.

Postconditions. Just ensuring that invariants are not left broken as a result of a card tear may not be enough to ensure that a method is correct. We may want to establish additional properties. For example, in our example in Fig. 1, we might want to ensure that if method `m` is interrupted by a card tear, it will either leave `p` unchanged of increase `p` by 2, and not say reset `p` to 0, which is allowed by the invariant; in this case we would like to establish `p == \old(p) || p == \old(p)+2` as postcondition of `m` in the event of a card tear.

There are two mechanisms that we can use to ensure that an invariant is not left broken (c.q. an additional postcondition is met) after a card tear occurs:

- the transaction mechanism; e.g., in Fig. 1, the transaction mechanism ensures that the invariant for `p` is maintained in the event of a card tear.
- the clearing of transient memory; e.g., in Fig. 1, the clearing of transient memory ensures that the invariant for `t[0]` is maintained (or, rather, re-established) in the event of a card tear.

The former mechanism is only relevant if an invariant (postcondition) involves transient data, the second mechanism is only relevant if it involves persistent data.

Given the nature of transient data, and the fact that transient data typically serves as scratch-pad memory, it is unlikely that we will be interested in any invariants or postconditions involving transient data. (Indeed, the whole idea of an invariant seems at odds with the notion of transient memory.) So, for many Java Card applications, it will not be necessary to take the clearing of transient memory into account to establish their correctness.

Invariants which only depend on persistent memory can be dealt without trying to formalize the transaction mechanism, in two ways:

– *Ensure that an invariant is never broken.*
  It may seem an overly simplistic approach, but in practice, many invariants are never broken. For example, in Fig. 1, the invariant `t !=null && t.length == 1` will never be broken. This is the approach taken in [1].
  Still, one has to be careful about the notion level of atomicity here. E.g., an invariant `a == b` will be temporarily broken during the execution of the statement `int x = (a++) + (b++)`, even though the invariant will hold before and after execution of the statement if there are no card tears. When reasoning at the level of source code our notion of atomicity will be coarser than what it really is.
– *Ensure that the invariant is never broken outside a transaction.*
  Some invariants will have to be temporarily broken. (E.g. if we are updating two fields and there is an invariant expressing a relationship between these fields, the invariant will typically be broken after updating the first of these fields.) If these invariant involves persistent data, then this should be done inside a transaction.

3 Modeling Card Tears

To model card tears inside Java we use the same trick used in [4,5], i.e. card tears are modeled as a special kind of exception, which can arise at any moment during execution. Like an exception, a card tear is effectively an abrupt change of the flow of control. A difference is that whereas an exception can be caught, a card tear cannot be caught, as there is no VM executing that could execute an exception handler. However, conceptually we can consider the recovery to a card tear that happens the next time the card powers up (i.e. the undoing of any unfinished transaction and the clearing of all transient data) as the exception handler for a card tear exception. We introduce a special exception class `CardTearException` for modeling card tears[3].

[3] Strictly speaking, `CardTearException` should not be an `Exception`, but rather an `Error`, because we clearly do not want `CardTearExceptions` to be caught by any existing `try-catch` blocks in a program. However, using `Error` would introduce a problem in JML.

3.1 Throwing a `CardTearException`

There are several ways to account for the possibility of a `CardTearException` being thrown at any moment during execution, namely at syntactical level, at semantic level, or at logical level:

a). One possibility is to do this purely syntactically, by desugaring any sequence of statements, e.g.

```
S1; S2;
```

to include calls to a method `possibleCardTear` before and after each statement, e.g.

```
possibleCardTear(); S1; possibleCardTear(); S2; possibleCardTear();
```

where `possibleCardTear` is a method which either performs a `skip`, or throws a `CardTearException`. We can even give a possible implementation of this method `possibleCardTear` in Java, for instance

```
possibleCardTear() {
  if (cardtear_counter-- < 0) throw new CardTearException();
}
```

where `cardtear_counter` is a global (i.e. final static) variable, initialized to an unknown value.
Such a syntactic approach has its limitations, namely the level of atomicity of statements that we can distinguish at the level of source code syntax. This notion of atomicity is coarser than it is in reality. E.g. in the example above we treat the statements `Si` as atomic, whereas in reality only individual byte code operations are atomic. For example, a statement such as `int x = (a++) + (b++)` would have to be rewritten into `a++; b++; int x=a+b;` if we want to include possible card tears after incrementing `a` or `b`[4].

b). Instead of modeling the possibility of card tears syntactically, as sketched above, an alternative would be to redefine our semantics of Java to include card tears. For instance, in the LOOP project we use a denotational semantics, and we could redefine the semantics of composition `;` and increment operation `++` to include the possibility of an exception being thrown. Effectively, this comes down to for instance changing the semantics of composition `;` to the composition of $\hat{;}$, where $S_1 \hat{;} S_2$ is defined as

$$possibleCardTear(); S_1; possibleCardTear(); S_2; possibleCardTear();$$

c). Another possibility of modeling card tears is at the logical level, i.e. in the logic used to reason about programs. For instance, if our reasoning about Java programs uses some Hoare logic, we could adapt all Hoare rules to allow for the possibility of card tears. Effectively, this comes down to for instance replacing the Hoare rule for composition `;` by the Hoare rule for $\hat{;}$.

4 Still, Java Card does not support the data types `double` and `long`, for which assignments are by definition non-atomic; see [8], section 17.4.

For the remainder of this paper, we leave it open which of the mechanisms above is used to model the possibility of card tears. Clearly, introducing explicit calls to `possibleCardTear()` at all program points quickly makes programs unreadable, so we prefer to leave the possibility of card tears being thrown implicit.

3.2 Specification and Verification Using `CardTearException`

Modeling card tears as exceptions is useful both when it comes to verifying and specifying Java Card code.

An `invariant` in JML has to hold if a method throws an exception. So an immediate consequence of modeling card tears as exceptions is that to verify a method we must ensure that invariants hold at every program point, as discussed earlier in Sect. 2 (and as in the approach of [1]).

Another advantage of treating card tears as exceptions is that it becomes possible to specify the behavior in the event of a card tear in JML, as mentioned as a wish in Sect. 1. This is not possible in the approach of [1]. For example, we could specify the behavior of the method `m` from Fig. 1 as follows:

```
//@ ensures p == \old(p)+2 && t[0] == \old(t[0])+2;
//@ signals (CardTearException) (p == \old(p) || p == \old(p)+2)
//@                                  && t[0] == 0;
void m() throws CardTearException{ ... }
```

Here the JML keyword `signals` is used to specify an *exceptional postcondition*, i.e. a condition that should hold after a certain exception occurs. Note that here we assume that the undoing of any unfinished transaction and the resetting of the transient memory occurs immediately after a card tear occurs, so that this has occurred before we exit the method.

Example 2 (specifications for arrayCopy(NonAtomic)). More example specifications that use the notion of `CardTearException` are given in Fig. 2. Here specifications are given for the Java Card API methods `arrayCopy` and `array-CopyNonAtomic`. These two methods are interesting examples because the only difference between them is what happens when a card tear occurs during their execution. The former method is atomic, so either all array entries are copied, or none are. The latter method is not atomic, so some array entries may be copied whereas others are not. The JML specifications in Fig. 2, more in particular the `signals` clauses, make this difference precise. Note that the specification of `arrayCopyNonAtomic` makes no assumptions on the order in which the array elements are copied.

4 Modeling the Clearing of Transient Memory

We now consider how to model the clearing of transient memory in the event of a card tear. Because transient data is completely unaffected by transactions, we can consider this issue in isolation, without taking into account how we model the transaction mechanism.

```
/*@   requires src != null && dest != null &&
  @            srcOff >= 0 && destOff >= 0 && length >= 0 &&
  @            srcOff+length <= src.length && destOff+length <= dest.length;
  @
  @ assignable dest[destOff..destOff+length-1];
  @
  @   ensures (\forall short i; 0 <= i && i < length
  @                          ; dest[destOff+i] == \old(src[srcOff+i]));
  @   signals (CardTearException)
  @              (\forall short i; 0 <= i && i < length
  @                              ; dest[destOff+i] == \old(src[srcOff+i]))
  @           || (\ forall short i; 0 <= i && i < length
  @                              ; dest[destOff+i] == \old(dest[destOff+i]));
  @*/
 native public static final short arrayCopy(byte[] src,
                                            short srcOff,
                                            byte[] dest,
                                            short destOff,
                                            short length);
/*@   requires ...
  @ assignable ...
  @   ensures ...
  @
  @   signals (CardTearException)
  @            (\forall short i; 0 <= i && i < length
  @                        ;    dest[destOff+i] == \old(src[srcOff+i])
  @                          || dest[destOff+i] == \old(dest[destOff+i]) );
  @*/
 native public static final short arrayCopyNonAtomic(byte[] src,
                                                     short srcOff,
                                                     byte[] dest,
                                                     short destOff,
                                                     short length);
```

Fig. 2. JML specifications for the API methods `arrayCopy` and `arrayCopyNonAtomic`. In the latter only the differences with the former are shown.

We model the clearing of transient memory by enclosing every method in a `try-catch`, where in the `catch` the transient memory is cleared, i.e. reset to the initial default for that type. For the code from Fig. 1, this desugaring results in:

```
  class A {
    // persistent field p, allocated in EEPROM
    byte p;
    //@ invariant p % 2 == 0 && 0<= p && p < 10;

    // transient array t, so t[0] is allocated in RAM
    byte[] t = makeTransientByteArray(1,CLEAR_ON_RESET);
    //@ invariant t != null && t.length == 1;
    //@ invariant t[0] % 2 == 0;

    //@ ensures p == \old(p)+2 && t[0] == \old(t[0])+2;
    void m() {
      try {
        beginTransaction();
        p++; t[0]++; p++;
        if (p < 10) commitTransaction();
               else abortTransaction();
        t[0]++;
      }
      catch (CardTearException e) {
        for (int i = 0; i < t.length; i++) t[i] = 0; // clear transient array t
        throw e; // re-throw the exception
      }
    }
  }
```

Note that in this particular example the explicit clearing of the transient array `t` in the event of a `CardTearException` will re-establish the invariant `t[0] % 2 == 0`.

One subtlety in the desugaring above is that during the clearing of transient memory in the `catch` block we do not want to allow card tears.

The only question in this desugaring is deciding which transient arrays a method should clear. The easiest way to decide this is to look at the postconditions (i.e. the `ensures` and `signals` clauses in conjunction with any `invariants`) that we want to prove for the method. Letting every method clear only the transient fields mentioned in its postconditions is sufficient. If a method calls another method, and a card tear occurs in this inner method call, this may lead to transient arrays being cleared several times, but as this clearing is clearly an idempotent operation, this is not a problem.

The only problem that can arise is when a specification refers to a transient field of another object to which the current object does not have access. In JML specifications the normal visibility constraints imposed by the Java modifiers (such as `private` or `protected`) can be loosened up, so it is possible for a JML specification of a method to mention a transient field `o.t[0]` of some other object `o` even though this field is not accessible from within that method. To cope with this, the object `o` in question would have to be extended to provide a method `clearTransients()` that clears its transient fields.

We should stress again that in Java Card applets transient data typically serves as scratch-pad memory, so that it is unlikely that we are interested in any invariants or postconditions involving transient data.

Note that both specifications in Fig. 2 exclude the effect of clearing transient memory: the `signals` clauses of `arrayCopy` and `arrayCopyNonAtomic` do not state that if `dest` or `src` are transient arrays their contents will have been cleared in the event of a card tear. We could modify the specifications to include this, by introducing a further case distinction in the `signals` clause on whether the arrays in question are transient or not, and including

```
(JCSystem.isTransient(dest) != JCSystem.NOT_A_TRANSIENT_OBJECT)
==>
 (\forall i; 0 <= i && i < dest.length; dest[i] == 0)
```

in the `signals` clause, and a similar statement for `src`. Here we use the Java Card API method `isTransient`, which can be use to test if an array is transient.

However, conceptually it is much more convenient not to make `arrayCopy` or `arrayCopyNonAtomic` responsible for clearing the arrays `src` and `dest` if they are transient, but to leave it up to the methods calling `arrayCopy(NonAtomic)`.

5 Modeling Transactions

We now turn to the issue of modeling transactions in Java. This comes down to the question of how conditional updates to persistent fields can be modeled in such a way that they can be undone in the event of an aborted transaction, caused by a card tear or by an invocation of `abortTransaction`. We do this by mimicking the way this can be implemented in hardware. Such an implementation involves some extra bookkeeping for persistent fields that are changed during a transaction. Two values will have to be recorded for these fields: the 'new', updated value, as well as the 'old' value the field had at the start of the transaction. There are roughly two strategies for doing this, as discussed in [9]. Suppose a persistent field `x` is modified during a transaction. One strategy, the *optimistic* strategy, is to log the old value of `x` at the beginning of the transaction, and use the logged value in the event of an aborted transaction to restore `x` to its original value. The other, *pessimistic*, strategy is to work on a temporary copy of a persistent field `x` during the transaction, and copy this updated version of `x` back to `x` when the transaction is committed. The optimistic approach entails some extra work in case the transaction is aborted, the pessimistic approach entails some extra work in case the transaction is committed. The Sun specification does not say anything which rules out or favors one of these two approaches. We will use the optimistic approach, introducing an extra 'backup' field x`bak` for every field x, but we could just as easily have used the pessimistic approach.

Below we show how the code given in Fig. 1 can be desugared to model the transaction in this way. As discussed in Sect. 3, we assume that at anytime the special `CardTearException` can be thrown.

```
class A {
  byte p = 0;
  byte pbak;                        // backup value of p
  //@ invariant p % 2 == 0 && 0 <= p && p <= 10;

  byte[] t = makeTransientByteArray(1,CLEAR_ON_RESET);
  //@ invariant t != null && t.length == 1;
  //@ invariant t[0] % 2 == 0;

  static boolean inTransaction = false;

  //@ ensures p == \old(p)+2;
  //@ signals (CardTearException) p == \old(p) || p == \old(p)+2;
  void m() {
   try {
        pbak = p;                   // backup p
        if (inTransaction) TransactionException.throwIt(IN_PROGRESS)
        inTransaction = true;       // begin transaction
        p++; t[0]++; p++;
        if (p < 10)
            inTransaction = false;  // commit transaction
        else {
            p = pbak;               // restore old value of p
            inTransaction = false;  // abort transaction
        }
        t[0]++;
    } catch (CardTearException e) {
      if (inTransaction) {
        p = pbak;                                      // restore old value of p
        for (int i = 0; i < t.length; i++) t[i] = 0;   // clear transient array t
        throw e;                                       // re-throw the exception
      }
    }
  }
}
```

The changes to the code are:

- An extra field `pbak` is introduced for the bookkeeping of the old value of `p` during a transaction.
- A static field (i.e. a global variable) `inTransaction` is introduced to record whether a transaction is in progress or not.
- The entire method is included in a `try-catch` construction, which, in case of a card tear, undoes the effects of any transaction if a transaction was in progress.
- Any calls to `begin-`, `commit-`, and `abortTransaction` are replaced by a code fragments which set `inTransaction`, and backup or restore the value of `p`.

In general, at the place where the `commit-` or `abortTransaction` we should check that a transaction is indeed in progress, or else throw a `TransactionException`, by including

```
if (!inTransaction) TransactionException.throwIt(NOT_IN_PROGRESS);
```

We have omitted this in the example above because it is obvious that here this situation does not arise.

One subtlety in the desugaring above is that during the bookkeeping associated with restoring an aborted transaction we should not allow card tears.

Although the example above is a very simple one, we believe that this desugaring of transactions can be used for most Java Card programs.

Similar to the modeling of the clearing of transient memory, the only difficult issue in this desugaring of transactions is deciding for which persistent fields should be restored. This issue can be solved in exactly the same way: only the persistent fields mentioned in the invariant and pre- and postconditions are relevant for the verification of an individual method, and only for these do we have to restore the old values in the event of a card tear, i.e. in `catch` block at the end of a method, and in the event of an `abortTransaction`. If a method calls another method, and a card tear occurs in this inner method call, this may lead to persistent fields being restored several times, but as this restoring is clearly an idempotent operation, this is not a problem.

Similar to the modeling of the clearing of transient memory, the only problem that can arise is when a specification refers to a persistent field of another object to which the current object does not have access. As we have mentioned before in JML specifications the normal visibility constraints imposed by the Java modifiers can be loosened up, so it is possible for a JML specification of a method to mention a persistent field `o.p` of some other object `o` even though this field is not accessible from within that method. To cope with this, the object `o` in question would have to be extended to provide methods `backupPersistents()` and `restorePersistents()` to backup and restore the values of its persistent fields.

6 Implementation

Figure 3 provides an overview of our project. We have described how the preprocessor can simulate some Java Card issues into a Java model. So far, we have not tried to mechanize the desugaring we have introduced in this paper.

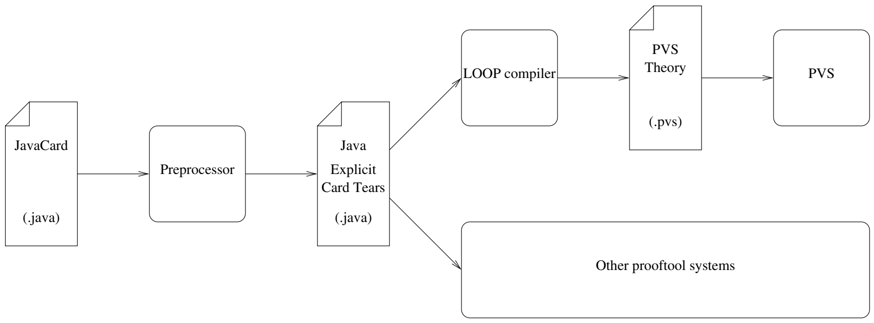


Fig. 3. Inbedding of Java Card transactions into Java based proof checking systems

It is quite easy to do the desugaring for the clearing of transient memory and for transactions by hand. The main question is how to deal with `possibleCardTear`. As discussed in Sect. 3.1, there are three ways of dealing with this. The last two of these, i.e. b) and c), require a large amount of work, namely redefining the semantics of all Java Card constructs, reformulating and reproving all Hoare rule, or redefining the whole weakest precondition strategy. In our LOOP/PVS setting the amount of work needed would be huge. Therefore it seems most practical to go for the syntactical approach a).

Furthermore, when verifying Java code in our LOOP/PVS setting, it seems wise to at least start with the more pragmatic approaches mentioned at the end of Sect. 2.1, and just verify that invariants are never broken outside transactions, i.e. the approach that is also taken in [1].

7 Remaining Issue

We believe that our model faithfully formalizes Sun's official specification of the transaction mechanism, with the possible exception of the issue discussed below.

One unclarity in Sun's specification of the Java Card API that we came across concerns the 'non-atomic' methods `arrayCopyNonAtomic` and `arrayFillNonAtomic` in the API class `javacard.framework.Util`. Indeed, as is often the case, the main value of our formalization may well be to reveal such potential ambiguities in the informal specification.

The API specification of these methods says that

"This method does not use the transaction facility during the copy operation even if a transaction is in progress. ..."

The precise meaning of this is not clear. For example, suppose we have two persistent arrays `p` and `q` and that during the execution of

```
p[0] = 0; q[0] = 2;
beginTransaction();
  p[0]++;
  arrayCopyNonAtomic(q,0,p,0,1);
endTransaction();
```

a card tear occurs immediately after the `arrayCopyNonAtomic`. In that case, it is not clear if `p[0]` will be restored to `0`, as our semantics would predict, or to `1`. In the former case all side-effects to `p[0]` during the transaction are undone, in the latter case the side-effect of `arrayCopyNonAtomic` is not undone. Experiments on actual smarts[5] confirms that the former happens. However, the quote above could be interpreted to mean that the latter should happen.

We do not have the space to discuss all the implications of this issue here, and, moreover, we are still investigating it, but we will make a report available about the outcome, so check our webpages for additional information.

8 Conclusion

We have shown how Java Card features such as card tears, transactions, and transient as opposed to persistent memory can be faithfully modeled inside Java, making it possible to use existing programming logics for Java to reason about these features. An advantage of the approach is that it is to a large extent independent of the Java semantics being used. An added benefit of such a model inside Java is that it is understandable to a larger audience – the desugarings should provide anyone with a good knowledge of Java with a clear understanding of the semantics of card tears and transactions, and with useful basis about reasoning about the features – and that we can use standard JML to specify properties of these features.

A disadvantage of the approach is that it is somewhat ad-hoc. A definition of a semantics for Java Card from scratch, be it denotational or operational semantics, or an axiomatic semantics as Hoare logic or weakest precondition calculus, would provide less ad-hoc and more rigorous semantics. Furthermore, it is somewhat unsatisfactory that we have to assume that during our event handling code there will not be another card tear.

It is important to realize that Java Card programs are extremely simple programs, without much complicated class hierarchies or OO structure to speak of. In many applications the only object, apart from some byte arrays used as fields, is a single object of class `javacard.framework.applet`. For such programs it is trivial to decide at compile time what the relevant persistent and transient data at runtime will be.

To our knowledge, the only other work on Java Card that does not ignore transactions and card tears is [1]. The approach presented there can be used to

[5] IBM JCOP 2.1.1 cards, 21id and 31bio to be precise.

prove that certain invariants are never broken, but cannot establish postconditions in the event of a card tear, as discussed in Sect. 2.1, or distinguish between the atomic and non-atomic API methods for copying arrays discussed in Ex. 2.

Unfortunately, the Java model is quite unreadable if it takes all explicit card tears into account. The implementation of the existing `TransactionExceptions` also really decreases readability, which makes this method less practical. On the other hand, knowing these syntactical problems it should not be too difficult to implement some of these solutions into the semantics of the proof system. For our own LOOP tool we are sure that this can be done.

Acknowledgment

Thanks to Joachim van den Berg of TNO-EIB for carrying out experiments on smart cards to confirm our own experiments to resolve the semantics of the 'non-atomic' methods discussed in Sect. 7.

References

1. Beckert, B., Mostowski, W.: A program logic for handling Java Card's transaction mechanism. In Pezzè, M., ed.: Fundamental Approaches to Software Engineering, FASE'2003. Volume 2621 of Lecture Notes in Computer Science. (2003) 246–260
2. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G., Leino, K., Poll, E.: An overview of JML tools and applications. In Arts, T., Fokkink, W., eds.: Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03). Volume 80 of Electronic Notes in Theoretical Computer Science (ENTCS)., Elsevier (2003) 73–89
3. Jacobs, B., Poll, E.: Java program verification at Nijmegen: Developments and perspective. Technical Report NIII-R0318, Dept. of Computer Science, Univiversity of Nijmegen (2003)
4. Hartel, P.H., de Jong Frz, E.K.: A programming and a modelling perspective on the evaluation of Java card implementations. In Attali, I., Jensen, T., eds.: 1st Java on Smart Cards: Programming and Security (e-Smart). Volume 2041 of LNCS., Springer-Verlag, Berlin (2000) 52–72
5. Poll, E., Hartel, P., de Jong, E.: A Java reference model of transacted memory for smart cards. In: Fifth Smart Card Research and Advanced Application Conference (CARDIS'2002), USENIX (2002) 75–86
6. Chen, Z.: Java Card Technology for Smart Cards. The Java Series. Addison-Wesley (2000)
7. Sun: Java Card 2.1 Runtime Environment (JCRE) Specification. Sun Micro systems Inc, Palo Alto, California (1999) `http://java.sun.com/products/javacard/`.
8. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Addison-Wesley (1996)
9. Oestreicher, M.: Transactions in Java Card. In: 15th Annual Computer Security Applications Conf. (ACSAC), Phoenix, Arizona, IEEE Comput. Soc, Los Alamitos, California (1999) 291–298 `http://www.acsac.org/1999/abstracts/thu-b-1500-marcus.html`.

Predictable Dynamic Plugin Systems

Robert Chatley, Susan Eisenbach, Jeff Kramer,
Jeff Magee, and Sebastian Uchitel

Dept of Computing, Imperial College London,
180 Queensgate, London, SW7 2AZ, UK
{rbc,sue,jk,jnm,su2}@doc.ic.ac.uk

Abstract. To be able to build systems by composing a variety of components dynamically, adding and removing as required, is desirable. Unfortunately systems with evolving architectures are prone to behaving in a surprising manner. In this paper we show how it is possible to generate a snapshot of the structure of a running application, and how this can be combined with behavioural specifications for components to check compatability and adherence to system properties. By modelling both the structure and the behaviour, before altering an existing system, we show how dynamic compositional systems may be put together in a predictable manner.

1 Introduction

There is a growing need for software systems to be extensible, as changes in requirements are discovered and fulfilled over time. In many cases it is inconvenient or costly to stop and restart an application in order to perform a change in configuration. By using a plugin architecture we can construct systems from combinations of components, with the architecture changing dynamically over time.

Szyperski describes components as units of composition that may be subject to composition by third parties [4]. Plugin architectures fit this description well. Plugins are components that can optionally be added to an existing system at runtime to extend its functionality. Each plugin may expose certain interfaces that it provides and requires [10]. By matching provisions to requirements, we can identify components that can be connected. By dynamically creating bindings between these components, calls can be made from a component requiring a service to another component that provides that service.

In an environment where systems can change through incremental addition and removal of components, it is desirable to be able to check for the preservation of properties as systems evolve. A group of components may be interacting correctly, but introducing a new component to the system may cause problems. Before adding new components, we would like to be able to ensure that undesirable behaviour will not occur, in order that configurations that might violate certain properties are not realised. Examples of such properties might be freedom from deadlock, liveness, or ensuring that an error state is never reached.

Our approach is to build and check a model that contains both structural and behavioural information.

The structural information consists of interfaces and bindings, which define sets of shared actions through which components can interact. However, they do not provide any information about the order in which these actions will be performed. This means that although we may be able to reason about the structure of systems of components based on this information, we are unable to reason in any way about their behaviour.

The behavioural information comes from the developer of a component, who can supply a specification of the way that component behaves (it is impossible to ascertain the programmer's intentions automatically). However, as components from different vendors can be combined in any number of different possible configurations, there is no way of writing a definitive model of how all different combinations will behave. To produce a model of the behaviour of the complete system requires composing the behavioural models for all of the components in a particular configuration in parallel, and ensuring that components are correctly synchronised where their interfaces are bound together. In this paper we show how such a model can be constructed, and hence the system's behaviour can be analysed.

As systems of plugin components can have components dynamically added (and removed) over time, and because one of the ideas of plugin components is to minimise the effort involved in configuring and administering a system, it is desirable that system models be generated and tested automatically. We show how our structural and behavioural specification techniques can be used for this, and how our tools can generate and analyse models automatically.

In the remainder of this paper we describe techniques for modelling the structure and behaviour of systems, and go on to discuss how the features of the plugin system map to the concepts used in these modelling techniques. We describe how models can be automatically generated from implemented components, present an example, and finally discuss related and future work.

2 Background

2.1 Plugin Framework

We have implemented a framework for plugin components, which we call MagicBeans, that can examine the compiled code of a Java component and automatically perform the matching and binding of interfaces at runtime [5]. The MagicBeans framework is more powerful than the plugin systems used to extend applications like web browsers, as we can handle plugins to plugins, creating arbitrarily complex configurations of components. An advantage over the plugin system used by Eclipse [13], is that we do not require that the system be restarted in order to pick up new plugins.

MagicBeans is implemented in Java, and allows a system to be composed from a set of components, each of which comprises a set of Java classes and other resources (such as graphics files) stored in a Jar archive. The MagicBeans platform is a component in its own right, but the system starts with it already in place as a bootstrap. The platform provides some static methods that can be called from any other component.

The platform maintains lists of all of the components in the system and the bindings between them. When a new plugin is added to the system, the platform searches through the classes and interfaces present in the new component's Jar file to determine how it can be connected to the components currently in the system. For each component, the plugin manager iterates through all of the classes contained inside the Jar file, checking for interfaces implemented (provisions) and methods accepting plugins of particular types (requirements), and compares these for compatability with the provisions and requirements of the other components currently in the system. To be compatible, a provision must be a subtype of a requirement. In each case that a match is found, the class implementing the provision is instantiated and a reference to the object created is passed to the other component, creating a binding.

2.2 Modelling Structure and Behaviour

Software Architecture describes the gross organisation of a system in terms of its components and their interactions. The Darwin ADL [10] can be used for specifying the structure of component based and distributed systems. Darwin describes a system in terms of components that manage the implementation of services. Components provide services to and require services from other components through ports. The structure of composite components and systems is specified through bindings between the services required and provided by different component instances. Darwin has both a textual and a complementary graphical form, with appropriate tool support.

Darwin structural descriptions can be used as a framework for behavioural analysis. Darwin has been designed to be sufficiently abstract as to support multiple views, two of which are the behavioural view (for behavioural analysis) and the service view (for construction). Each view is an elaboration of the basic structural view: the skeleton upon which we hang the flesh of behavioural specification or service implementation.

Focussing on the behavioural view, we can use simple process algebra - Finite State Processes (FSP) [9] - to specify behaviour. A complete system specification can be written by using the same action names in the behavioural specification as in the Darwin service descriptions. These specifications are translated into Labelled Transition Systems (LTS) for analysis purposes. Analysis is supported by the Labelled Transition System Analyser (LTSA) tool.

3 Generating a Model of the System

In our system of plugin components, the runtime plugin framework (MagicBeans) forms a middleware platform. This is responsible for initialising all of the components, matching the required and provided interfaces of the new component against those of the other components already in the system, and creating bindings between them, in order to create an application. Any addition or removal of components has to be done via the plugin framework. The framework therefore has information about all of the components currently in the system, and how they are connected.

The framework can use this information to produce a textual specification in Darwin that gives a snapshot of the current system configuration. This can be done at runtime, based solely on the information present in the compiled code of the components and the current state of the system. There is no need for the developer to provide Darwin descriptions of each component, as these can be generated automatically from the bytecode.

3.1 Matching Plugin Concepts with Darwin Concepts

Our plugin components comprise collections of (Java) classes and interfaces bundled together in a Jar file (which may also contain other resources such as graphics or data files). Below is the Java code for a basic filter plugin, and the corresponding Darwin description that is generated from it. The Java code follows an outline that would be the same for any plugin. The code for the class and interface would be compiled and packed into a Jar file, forming the plugin component.

For each Jar file we will have a corresponding `component` construct in Darwin. The Jar file may contain a number of class files representing interfaces. These are collections of methods that define types. We equate them with Darwin interface definitions which perform the same function.

```
Java :

        public interface Filter { public void data( String x ); }

        public class FilterImpl implements Filter {

             Filter next;

             // constructor
             public FilterImpl() {
                 PluginManager.register( this );
             }

             // implementation of Filter interface
             public void data( String x ) {
                 if ( next != null ) { next.data( x ); }
             }

             // method to be called by plugin platform
             public void pluginAdded( Filter f ) { next = f; }
         }
```

```
Darwin :

         interface Filter { data; }

         component FilterImpl {

              require next:Filter;
              provide Filter;
         }
```

Some of the classes in the Jar file may be declared as implementing certain public interfaces. These are classes that provide services that can be used by other components. The inclusion of such a class in a plugin is equivalent to declaring a Darwin component to have a provided port with the type named by the interface. Such a class may be instantiated several times, by a third party, to produce objects that provide this service. We do not have explicit names for these objects, so in Darwin we just declare the type of the provided port.

Components can use services provided by other components. When a new plugin is added to a system, the component that accepts it needs to be able to call methods provided by that plugin in order to use it. In Darwin this corresponds to a required port. In Java we need a reference to an object of a certain type in order to be able to call its methods. The mechanism by which we acquire such a reference in the plugin system is as follows.

An object registers as an observer with the plugin platform, by calling a static method `register()` in the PluginManager class. To be notified of new plugins, the object can define a number of `pluginAdded()` methods with different parameter types. When a new plugin is connected the platform picks the relevant method and calls it, passing a reference to the object from the new component that provides the service. In the body of the `pluginAdded()` method this reference is assigned to a field of the appropriate type.

We generate requires ports in the Darwin specification for any field in the Java which is assigned to within the body of one of the `pluginAdded()` methods. We name the port with the name of the field as declared in the class, `next` in the above example. Naming required ports is necessary as it is possible for a component to have more than one required port of the same type, as a component may accept multiple plugins of the same type. These could be assigned to different fields in the class, or added to an array. For example, a forking filter would forward data to two different downstream components, and so would accept, and keep references to, two plugins with the same interface.

It should be noted that it is much more difficult to extract information about the required services from a component than the provided service. We have to look for names of fields, and examine the body of the `pluginAdded()` method by processing the Java bytecode, rather than simply finding the type of the class. It is a trait of object-oriented programming that objects typically declare the methods that they provide, but not those that they use from other objects.

When constructing a system, we work at the level of components. A new Jar file is loaded to add a component. Any provided ports in the new component that match required ports in other components, or vice versa, are identified. The class that provides the service is instantiated by the plugin platform and any observers in the component requiring the service are notified, passing a reference to this new object. This process creates a binding between the two components. In Darwin terms, we model the complete system as a component, and add to it an instance of the providing component, which is given an arbitrary, but unique, name. We also add a binding between the relevant ports and components. The following would be generated for a chain of two filters:

```
component System {

        inst  f:FilterImpl;
             f2:FilterImpl;

        bind f.next -- f2.Filter;
}
```

3.2 Specifying Behaviour

A simple example showing how these concepts might be extended to include behaviour (a specification of the order in which actions are performed) is a client connected to an email server. The Client component contains an interface Email, declaring the methods `login()`, `fetchMail()` and `sendMail()`, and when notified of an object of this type will call these methods. The Server component contains a class that implements the `Email` interface. The plugin framework can create a description of the interface and the two components in Darwin. Provided and required ports are declared with the appropriate types. In the example, the system as a whole comprises one instance each of the Client and Server components, with the two ports connected by a binding.

```
interface Email { login; fetchMail; sendMail; }

component Server {

     provide Email;
}

component Client {

     require serv:Email;
}

component System {

     inst s:Server;
          c:Client;

     bind c.serv -- s.Email;
}
```

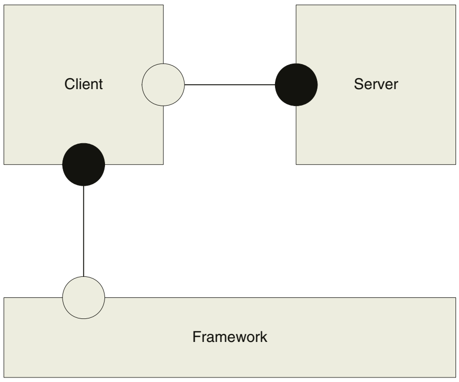


Fig. 1. Client provides FSPDefinition to the plugin framework.

The information in the Darwin description is purely structural. In order to add some information about the behaviour of each of the components, we need a way of including an abstract description of the behaviour with the component. We do not want to provide the behavioural model separately from the component as one of the ideas underpinning plugin technologies is that plugins should be deployed as single entities that include everything they need in order to be used.

The approach we have taken is to allow each component to have a another provides port where it can provide a textual description of its behaviour (in FSP) as a string. A binding can be made between this port and a requires port on the plugin framework (which is itself a component that can be connected in the same way that any other in the system can), see Figure 1.

When the framework is constructing the Darwin to describe the current state of the system, it will request the FSP from any components that provide it, and include this in the model. For example, we can include the following (Java) class in the Client component, allowing it to provide an FSP description of its behaviour:

```
public class ClientBehaviour implements FSPDefinition {

    public String getFSP() {

        return ‘‘Client = ( serv.login -> serv.fetchMail
                            -> serv.sendMail -> Client ).’’;
    }
}
```

When the framework generates the system description, it requests the FSP description from the Client and includes it in the Darwin inside the definition of the Client component (inside a special type of comment `/% ... %/`) in the Darwin specification, as below. The behavioural description shows an ordering of actions called through the `serv` port (the client logs in, then fetches email, then sends email), which cannot be derived from the interface descriptions alone.

In the case that a component does not provide an FSP description of its behaviour, as with the Server component in this example, we generate a process that allows any of the actions from the component's provided interfaces to be performed in any order.

```
interface Email { login; fetchMail; sendMail; }
interface FSPDefinition { getFSP; }

component Client {

     require serv:Email;
     provide FSPDefinition;

     /% Client = ( serv.login -> serv.fetchMail
                   -> serv.sendMail -> Client ). %/
}

component Server {

     provide Email;

     /% Server = ( { login, fetchMail, sendMail } -> Server ). %/
}

component Framework {

     require fsp:FSPDefinition;
}

component System {

     inst s:Server;
          c:Client;

     bind c.serv -- s.Email;
          f.fsp  -- c.FSPDefinition;
}
```

Here we have included the Framework component in the model, and show how the Client provides the behavioural definition through a port which is bound to the corresponding port in the Framework. We have omitted any description of the behaviour of the Framework, as it is part of the infrastructure rather than a user component. We assume that the Framework is transparent and will not introduce any behavioural problems.

3.3 Changes of Configuration

As the configuration of a piece of software constructed from plugin components changes over time, the way that particular components behave may also change. Components may behave differently depending on whether they have other components connected to their required ports.

When a plugin is connected to the system, other components need to change their behaviour to take advantage of the new services provided. Existing components need to be notified that a new component has been connected. To achieve this, components register with the plugin framework as observers, to be notified when a change in configuration occurs that is relevant to them.

The framework calls the observer back through the `pluginAdded()` method. In the FSP we can use the corresponding action `pluginAdded` action as a signal to change from one mode of behaviour to another. If a component implemented a basic matrix analysis algorithm, but allowed a plugin to be connected that provided a more efficient implementation of this algorithm, the component might perform the calculation itself while its requires port is unbound. If and when it is notified that a plugin has been added (the port has been bound), the component will change its behaviour so that from then on the call is delegated to the plugin. This could be described in FSP as follows:

```
component MatrixSolver {

    require fast:Algorithm;

    /%
    MatrixSolver = ( input -> calculate -> output -> MatrixSolver
                   | pluginAdded -> FastSolve ),
    FastSolve    = ( input -> fast.solve -> output -> FastSolve ).
    %/
}
```

3.4 Specifying Properties

In FSP, safety and liveness properties can be specified for a model, and we can check these using a model checker. We also have the facility for expressing properties in a linear temporal logic. Currently we manually specify properties textually in the tool, but we anticipate that properties could be provided in components in the same way that behavioural specifications are. Properties could then either be provided as plugins in their own right, to plug in to the platform, or be integrated into other components. The platform could then incorporate them into the model.

3.5 Composing the System

The Darwin compiler constructs a parallel composition of the behaviours of each of the separate components, employing an appropriate relabelling such that components that are bound together are synchronised. For every pair of ports that are bound, `providesport.action` is relabelled to `requiresport.action`. Any action included in a behavioural description that is not part of one of the interfaces of one the ports of a component is treated as an internal action. The resulting FSP model can be compiled to a labelled transition system and checked for properties such as deadlock [9].

When a new component is identified for addition to the system, we can determine how we intend to bind the new component, and then build a model of how the system would behave if the new component were connected in that way. If we do this before the component is connected, we can use the model to check whether adding the component will cause the system to violate any properties that we wish to hold. This information can be used to decide whether or not a new component should be bound to the system in a certain way, or added at all.

4 Predicting Behaviour

We consider an example based loosely on the Compressing Proxy Problem [7]. A set of components are chained together to form a pipeline through which data can flow. We will allow further components to be plugged in to the end of the pipeline increasing the length of the chain over time. The basic premise of the problem is that in order to increase the efficiency of data transfer along the pipeline, a compression module is introduced at either end, compressing the datastream at the source and decompressing it again at the sink.

In the original Compressing Proxy Problem, the pipeline comprises a set of filters which all run in a single UNIX process. Integrating a compression module that uses gzip with this system requires some thought, as gzip runs in a separate process. An adapter is therefore used to coordinate the components. . In this example we consider only the source end of this situation, although similar issues are involved at the sink.

Each component is implemented as a set of Java classes and interfaces packaged into a Jar file. We start the plugin framework, with a Source as the component that forms the core of the system. The Source generates data and sends it down stream, see Figure 2. Each component provides an FSP definition of its behaviour. The Source component includes the following class:

```
public class SourceBehaviour implements FSPDefinition {

    public String getFSP() {

        return ‘‘Source = ( next.data -> Source ).’’;
    }
}
```

Each of the other component Jar files contains a similar class. We add a plain Filter to the pipeline. The `Filter` simply reads data from upstream and passes it on downstream. `FilterImpl` implements the `Filter` interface, and has a field of type `Filter` for a reference to the next component in the pipeline. `FilterImpl`'s `data()` method just calls the next component's `data()` method. This is specified in FSP as:

```
FilterImpl = ( data -> next.data -> FilterImpl ).
```

The gzip compressor cannot be placed directly into the pipeline, and so needs an adapter component to pass data to it. The GZip component then plugs in to

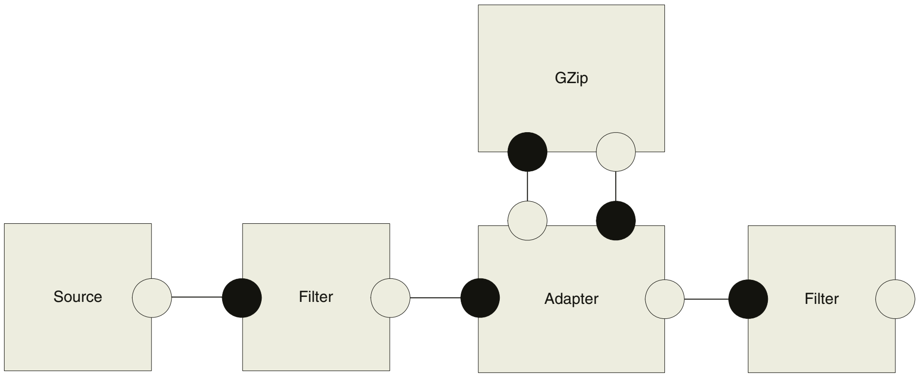


Fig. 2. Arrangement of components in pipeline with gzip.

the adapter. When there is no gzip processor present, the adapter should behave like a plain filter. When in adapting mode, the adapter sends packets out to the processor and reads the processor's output back in before sending the processed packets on downstream. The `pluginAdded` action triggers the transition from plain filter to adapting behaviour.

```
Adapter     = FilterImpl,
FilterImpl  = ( data -> next.data -> FilterImpl
              | pluginAdded -> Adapt
              ),
Adapt       = ( data -> out.packet -> ToProc ),
ToProc      = ( out.packet -> ToProc | out.end -> FromProc ),
FromProc    = ( packet -> FromProc | end -> next.data -> Adapt ).
```

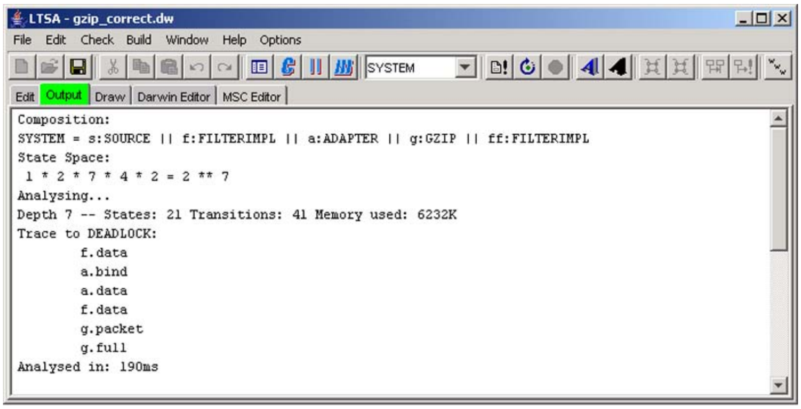
The complete Darwin and FSP specifications can be found at `http://www.doc.ic.ac.uk/~rbc/writings/fase04_appendix.pdf`. Using the Darwin compiler to translate this specification to FSP, then compiling that to an LTS model, enables the use of a model checker to perform a check for deadlock.

If we generate the model for the system without the gzip processor, then we can check the behaviour of the basic pipeline. In order to ensure that the adapter does not enter its adapting mode, it needs to be prohibited from performing the `pluginAdded` action. This can be done by composing the system in parallel with a process modelling the framework that synchronises on `a.pluginAdded` but never performs this action. Such a process can be defined as `STOP` with an alphabet extension to include the `a.pluginAdded` action.

```
Framework = STOP + {a.pluginAdded}.

||NoGZip = (System || Framework).
```

If we build the NoGZip process and check it, the model checker reports that it is deadlock free. If we add the GZip component, remove the constraint


LTSA - gzip_correct.dw
File Edit Check Build Window Help Options
SYSTEM
Edit Output Draw Darwin Editor MSC Editor

```
Composition:
SYSTEM = s:SOURCE || f:FILTERIMPL || a:ADAPTER || g:GZIP || ff:FILTERIMPL
State Space:
 1 * 2 * 7 * 4 * 2 = 2 ** 7
Analysing...
Depth 7 -- States: 21 Transitions: 41 Memory used: 6232K
Trace to DEADLOCK:
        f.data
        a.bind
        a.data
        f.data
        g.packet
        g.full
Analysed in: 190ms
```

Fig. 3. Screenshot from LTSA showing trace to deadlock.

so that `pluginAdded` can occur, rebuild the model and check again, we find that the following trace leads to a deadlock: `f.data, a.pluginAdded, a.data, f.data, g.packet, g.full` (as shown in Figure 3). This indicates that adding the gzip processor can lead to a deadlock if GZip's output buffer becomes full before the adapter is ready to accept output from the gzip processor.

To have the system work correctly without deadlocking requires replacing the adapter component with one that will accept output from the processor before having sent it the `end` signal to say that the input has finished, or using a GZip component that never tries to write any output before it receives the end of input signal, i.e. it has infinite capacity buffers.

5 Tool Support

The Labelled Transition System Analyser is a tool that compiles FSP into LTS models and checks properties on those models [9]. The LTSA itself now uses our plugin architecture, and we have developed a plugin for it to allow Darwin to be written and translated to FSP. We have also added an extension where the LTSA hosts a server that listens for Darwin specifications which are sent to it over the network.

Using these extensions, we can run a plugin application on one machine, and whenever a change is about to be made to the configuration, generate a Darwin/FSP specification and send it over the network to an instance of LTSA running on another machine. The LTSA then builds and checks the model. Producing a model of what the system would be like when a component is added before actually commiting and making the bindings, we can use the model checker to decide whether or not adding that component and making the proposed bindings is a safe thing to do.

We could use this process to check a set of possible bindings to see if any are unacceptable because of violation of properties. Running the checks on a remote machine means that we do not have to include all of the code for the model checker in the plugin platform.

Building and checking an LTS model can be expensive in terms of computation. Depending on the frequency with which changes in the system configuration are made, and how sure the administrators need to be that the resulting system will not deadlock, it may or may not be worth doing. Checking new additions to a large business server, where changes are large, infrequent, but business critical could definitely be justified. Checking the correctness of configurations of a desktop music player application when trying out different GUI components might well not be.

6 Related Work

There is a general movement towards the idea that the specification of a component should include information about its behaviour as well as its interface [2]. Several ADLs have been extended or complemented with languages for describing behaviour, for example C2SADEL [11] which uses logic to specify behaviour, or Wright [6] and PADL [3] which use process algebra.

The idea of incorporating the specification with the component is supported by Microsoft's AsmL [1]. This allows for the runtime verification of the behaviour of the implementation against the specification.

Another angle on including within a component a way to check that it meets some property is the use of proof-carrying code [12]. Components can be provided along with a proof that they fulfil some property. The system on which they are intended to run can verify these proofs using a proof checker.

The Bandera project [14] aims to extract process models directly from Java code, so that models can be built and checked directly, without human intervention.

In the work that we have presented here, we combine the behavioural descriptions for all components and check for a property. It might be interesting to see whether it is possible to use techniques designed for finding the assumptions necessary for assume-guarantee reasoning [8] to find an assumption that must hold for a component being added to the system, and check that component against the assumption separately from the system.

7 Conclusions and Future Work

We have presented a technique for automatically generating a description of the structure and behaviour of an application that has been composed dynamically from plugin components. Using tools we can compile this description to an LTS model, which we can test, using a model checker, to determine whether various desirable system properties hold.

The structural description can be generated automatically by the plugin middleware, based on the interfaces exported by each of the plugins and the bindings made between them. Behavioural information for each plugin is given in the form of a description in the FSP process calculus which is included in the component. By combining the behavioural information about each component with the description of the system structure, a model of the behaviour of the system as a whole can be generated.

The model can be compiled to the form of an LTS which can be analysed automatically, using a model checker, for adherence to desired system properties. Performing such analysis before a new plugin as added to the system allows us to predict whether the addition of this new component would cause the system to behave in an undesirable way.

Future work in this area could include trying to extract more behavioural information directly from the code of the components, rather than requiring the developer to write the specification by hand. Some techniques for doing this are being developed as part of the Bandera project [14] which could possibly be used. This could allow behavioural specifications to be generated automatically, rather than requiring the developer to write them in a language that may well be unfamiliar. However, if the model that is generated is too detailed then we may suffer from the state explosion problem when model-checking. Another approach would be to produce tools to assist developers in writing the behavioural specifications.

With our current technology, plugin systems are constructed by matching port types, and the techniques discussed here can be used to check the resulting system for adherence to a property. The use of behavioural properties could be extended to further direct and constrain the construction and reconfiguration of systems beyond what is currently possible.

Acknowledgements

We would like to acknowledge our colleagues in the Distributed Software Engineering and the SLURP group at Imperial for their participation in the discussions that helped to refine this work. We would also like to acknowledge the support of the European Union under grant STATUS (IST-2001-32298) and the EPSRC under grant READS (GR/S03270/01).

References

1. M. Barnett, W. Grieskamp, C. Kerer, W. Schulte, C. Szyperski, N. Tillmann, and A. Watson. Serious specification for composing components. In *6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction*, 2003.
2. M. Barnett and W. Schulte. The ABCs of specification: AsmL, behavior, and components. *Informatica*, 25(4):517–526, Nov. 2001.
3. M. Bernardo, P. Ciancarini, and L. Donatiello. Architecting families of software systems with process algebras. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(4):386–426, 2002.
4. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Pub Co, 1997.

5. R. Chatley, S. Eisenbach, and J. Magee. Painless Plugins. Technical report, Imperial College London, www.doc.ic.ac.uk/∼rbc/writings/pp.pdf, 2003.
6. D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 175–188, 1994.
7. D. Garlan, D. Kindred, and J. Wing. Interoperability: Sample Problems and Solutions. Technical report, Carnegie Mellon University, Pittsburgh, 1995.
8. J. Cobleigh, D. Giannakopoulou and C. Pasareanu. Learning Assumptions for Compostional Verification. In *Proc. of TACAS 2003.* LNCS, April 2003.
9. J. Magee and J. Kramer. *Concurrency – State Models and Java Programs.* John Wiley & Sons, 1999.
10. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proceedings of the 5th European Conference on Software Engineering, Sitges, Spain, 1995*, pages 137–154. Springer Verlag, 1995.
11. N. Medvidovic, D. Rosenblum, and R. Taylor. A language and environment for architecture-based software development and evolution. In *ICSE '99*, 1999.
12. G. C. Necula and P. Lee. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, Jan. 1997.
13. Object Technology International, Inc. Eclipse Platform Technical Overview. Technical report, IBM, www.eclipse.org/whitepapers/eclipse-overview.pdf, July 2001.
14. O. Tkachuk, M. B. Dwyer, and C. Pasareanu. Automated environment generation for software model checking. In *Automated Software Engineering*, 2003.

A Correlation Framework for the CORBA Component Model

Georg Jung, John Hatcliff, and Venkatesh Prasad Ranganath

Department of Computing and Information Sciences, Kansas State University
234 Nichols Hall, Manhattan, KS 66502, USA
{jung,hatcliff,rvprasad}@cis.ksu.edu

Abstract. Large distributed systems, including real-time embedded systems, are increasingly being built using sophisticated middleware frameworks. Communication in such systems is often realized using in terms of asynchronous events whose propagation is implemented by an underlying publish/subscribe service that hooks components into a generic event communication channel. *Event correlation* – a mechanism for monitoring and filtering events – has been introduced in some of these systems as an effective technique for reducing network traffic and computation time. Unfortunately, even though event correlation is used heavily in frameworks such as ACE/TAO's real-time event-channel and in mission critical contexts such as Boeing's Bold Stroke avionics middleware, the industry standard CORBA Component Model (CCM) does not include a specification of event correlation. While previous proposals for event correlation usually offer sophisticated facilities to detect combinations in the stream of incoming events, they have not been constructed to fit within the CCM type system, and they offer relatively little support for transforming and rearranging filtered events into meaningful output events. In this paper, we present the design rationale, syntax, and semantics for a new and highly flexible model for event correlation that is designed for integration into the CCM type system. Our model has been integrated and tested in the Cadena development and analysis framework, which has been designed to support development of mission-control applications in the Boeing Bold Stroke framework.

1 Introduction

As software systems become more distributed, developers are increasingly turning to component-based development frameworks such as Java Enterprise Beans (EJB) and the CORBA Component Model (CCM) to manage the complexities associated with building and deploying distributed systems. A major advantage of such component based systems working on sophisticated middleware in general is the clear separation of concerns, which distinctly isolates the stages of the development process as well as it divides business logic from infrastructure, allowing to synthesize substantial parts of the implementation directly from the specification. Further, CCM as an established industry standard based on CORBA, introduces system independence and a high level of interoperablity into the development process. As a result, component-based development is being explored in real-time safety/mission-critical domains as a mechanism for incorporating non-functional aspects such as real-time, quality-of-service, and distribution, permitting the developer to focus on application-specific parts.

Communication between components in such systems is often phrased in terms of asynchronous events whose propagation is implemented by an underlying publish/subscribe service that hooks components into a generic event communication channel. In these event services, *event correlation* – a mechanism for monitoring and filtering events – often plays a crucial role in reducing network traffic and computation time.

To illustrate, consider the common case where one component C receives events a and b from components A and B, respectively, and generates a new output event c which is synthesized in some way from a and b – specifically, C requires *both* a and b before it can generate its own output event. If component A issues events at a higher frequency than component B (as is often the case in real-time periodic systems), many a events will be discarded by C as it awaits for an accompanying b. Obviously a communication channel which is able to filter out such additional events saves logic and computation time in the receiving component and reduces network traffic in the system. Specifically, we would like the event communication layer itself to monitor the event flow and forward an a and b together to C only when both events have occurred. Depending on the complexity of communication this improvement is often substantial.

Unfortunately, even though event correlation is used heavily in frameworks such as ACE/TAO's real-time event-channel and in mission critical contexts such as Boeing's Bold Stroke avionics middleware, the CCM specification does not include a specification of event correlation. While previous proposals for event correlation usually offer sophisticated facilities to detect combinations in the stream of incoming events, they have not been constructed to fit within the CCM type system, and they offer relatively little support for transforming and rearranging filtered events into meaningful output events.

In this paper, we present a new and highly flexible model for event correlation that is simple in syntax and rich enough in features to specify complicated correlations. Increased flexibility is achieved by splitting an event correlator into two phases: first, a *filter phase* monitors the event flow for the desired event combination, then a second closely-interacting *transformation phase* disassembles input events and reassembles payloads from the input events into output events in a programmable manner. The splitting of correlation into these two phases (specifically, the introduction of the programmable transformation facility) allows our correlation framework to be tightly integrated with the event type system of CCM.

The contributions of the paper are as follows.

- We present a novel event correlation framework that decomposes event correlators into event filter and event transformer stages.
- We define a formal semantics for this correlation framework in terms of a language over event sequences.

- We show how the notion of event transformer allows for the first time a correlation framework to be incorporated into CCM and integrated with the CCM event type system.
- We describe how our correlation framework is implemented in the Cadena environment for development of high-assurance distributed systems, and how correlation specifications given at the modeling level are translated to implementations in the underlying middleware layers.
- We illustrate how our framework can be used to correlation problems that are representative of those in avionics applications but are more difficult to solve using previous frameworks.

These results remove barriers that previously prevented applications that relied heavily on correlation (such as those built in Boeing's Bold Stroke program) from being transitioned to a CCM framework where standardization and a richer deployment framework provide a variety of benefits.

The rest of this paper is organized as follows. Section 2 presents the syntax and semantics of event filter expressions Section 3 describes the event transformers. Section 4 illustrates how our correlation framework can be used to implement dynamic changes to correlations. Section 6 discusses related work, and Section 7 concludes.

2 Syntax and Semantics of Filter Expressions

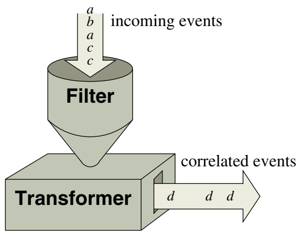
2.1 The Filter Syntax

Previous approaches build on atomic expressions which access the payload of an event and associate truth values according to whether the event with its attributes satisfies the atomic expression. While our model in general is independent from the actual form of the underlying atomic expressions, we chose to only consider the arrival of an event, since this strategy follows the concept of the CCM architecture in the sense that we connect typed source ports to typed sink ports from known entities, allowing the communication channel to have knowledge about the connections but leaving any assessment of the payload values other than rearrangement to the components of the system. The time of the issuing of the event (e. g. represented in the form of a timestamp attached to the event) as well as its source and its type are considered intrinsic properties of the event and hence visible to the correlator. Note that knowledge of the generation time of an event is implicitly assumed by all above mentioned previous works on event correlation [6, 7, 12, 10], otherwise the use of a sequential operator (see below) is infeasible. Accordingly, we can assign a single identifier to every source port connected to a correlator.

In the reminder of this section, we will assume that there are source components A, B, C, ... connected to the Event Channel (i. e. the communication channel provided by the middleware), and that these components issue the events a, b, c, ... with the types τ_a, τ_b, τ_c, ... respectively. As mentioned above, the payload of an event is not accessible to the filter, thus we can identify a finite set

$$\begin{aligned}
\mathsf{filter} &::= \mathsf{sequence}\ (\ \mathsf{||}\ \mathsf{filter}\ )^{*}\\
\mathsf{sequence} &::= \mathsf{collection}\ (\ ;\ \mathsf{sequence}\ )^{*}\\
\mathsf{collection} &::= \mathsf{accumulation}\ (\ \mathsf{|}\ \mathsf{collection}\ )^{*}\\
\mathsf{accumulation} &::= \mathsf{atom}\ (\ +\ \mathsf{accumulation}\ )^{*}\\
\mathsf{atom} &::= (\mathit{label}\,\mathsf{:})^{?}\ \mathit{event}\\
&\quad\ \mid (\mathit{label}\,\mathsf{:})^{?}\ (\ \mathsf{sequence}\ )
\end{aligned}$$

(a) Filter Expression Grammar



(b) Filter/Transformer

Fig. 1. Filter Grammar and Role.

$\Sigma = \{a, b, c, \ldots\}$ of possible events occurring in an infinite sequence as input of the correlator by considering two events equal iff they come from the same source port. Similar to the other mentioned approaches, we use an expression which we call the *filter expression* to denote the subsequences which are of interest for the receiver of the correlated event.

The syntax of the filter expressions is closely related to that of previous approaches such as e. g. the ECL expressions in [10], the Event Composition Operators in [7] or the Policy language in [6]. It is further based on our assessment of the Boeing's Bold Stroke and SAE AADL (Avionics Architecture Description Language[1]) frameworks. In this approach we present three basic combinators and a parallel combinator. Informally the three combinators are (1) the *accumulation* of events, i. e. both of two events a and b have to occur, regardless of the order (written a+b), (2) the *collection* of events, i. e. at least one of two events a and b has to occur (written $a\,|\,b$), and (3) the *sequence* of events, i. e. of two events a and b both have to arrive in the given order (written $a\,;b$). An abbreviated grammar of the filter expressions is given in Fig. 1(a). In this grammar, "$(\ldots)^{*}$" stands for zero or more and "$(\ldots)^{?}$" for zero or one instance of the item given inside the parenthesis. Note that all combinators are defined with arity two. While this does not impact the expressiveness of the combinators since they are associative [5], it greatly simplifies the formal definition of dynamic semantics given in section 4. Further, note that our approach enabled us to reduce the number of different combinators as compared to previous models without loosing flexibility. In fact, we believe that in our approach the expressions are more intuitive instead.

2.2 Semantics of the Three Basic Combinators

We now define the semantics of filter expressions in a way similar to regular expressions. This approach provides a firm formal background while leaving the computational model for implementing a filter open to the choice of the programmer. Essential is the concept of a *match*, which we first define for a basic atomic expression composed of a single event:

[1] See `http://www.sae.org/technicalcommittees/aasd.htm`.

Definition 1. *A sequence* $\boldsymbol{s} = e_1 \dots e_n$ *of events* $e_1, \dots, e_n \in \Sigma$ **matches** *a singleton filter expression* a*, iff an event* a *is in* $\boldsymbol{s}$*, i. e. there is an* i *with* $e_i = a$*.*

Next, we define the basic combinators `+`, `|` and `;`. The parallel operator `||` is discussed later.

Definition 2. *A sequence* $\boldsymbol{s}$ *of events* **matches** *a filter expression* $x_1 \oplus x_2$ *iff*

– $\oplus$ *is* `+` *and* $\boldsymbol{s}$ *matches* x_1 and $\boldsymbol{s}$ *matches* x_2*.*
– $\oplus$ *is* `|` *and* $\boldsymbol{s}$ *matches* x_1 or $\boldsymbol{s}$ *matches* x_2*.*
– $\oplus$ *is* `;` *and the sequence* $\boldsymbol{s}$ *can be split into two subsequences* $\boldsymbol{s_1}$ *and* $\boldsymbol{s_2}$ *such that* $\boldsymbol{s} = \boldsymbol{s_1} \cdot \boldsymbol{s_2}$ *and* $\boldsymbol{s_1}$ *matches* x_1 *and* $\boldsymbol{s}_2$ *matches* x_2*.*

To illustrate, consider the expression a+b and a;b. While the former is matched by the sequence *bbca*, the latter is not. Other matched expressions are e. g. b|d or b;a. Note that “+” and “|” are commutative, while “;” is not [5]. We will call the set $\mathrm{M}(x) = \{\boldsymbol{s} \in \Sigma^* \mid \boldsymbol{s} \text{ matches } x\}$ the set of matches of the expression x. Clearly, there are infinite sequences in the set of matches of an expression x. Nevertheless for the filter we are only interested in a notification whenever a match first is complete.

Definition 3. *A* **shortest match** *of an expression* x *is a sequence* $\boldsymbol{s}$ *of events such that* $\boldsymbol{s}$ *matches* x *but no proper prefix of* $\boldsymbol{s}$ *matches* x*.*

In analogy to regular languages we call the set of shortest matches of an expression x the language $\mathrm{L}(x)$ of x.

For example, *aab*, *aaab* and *aaaaaab* are shortest matches for the expression b with $a, b \in \Sigma$, *aaba* is a match, but not a shortest match. Shortest matches for the expression a+b|a+c are e. g. *ba*, *ac*, *bbcbca*, *aac*.

Sequences we can define with these three basic combinators are a subset of regular languages ([5]), hence it is obvious that we can construct an acceptor.

2.3 The Trigger

Definition 2 implies that for the combinators `+` and `|` both subexpressions have to be checked on the same sequence $\boldsymbol{s}$, which means that their acceptors are both executed in parallel. Consider the sequence *abbabaa*... and the expression a;b;a. Then there are two successive shortest matches in the sequence

$$\underbrace{abba}\,baa\dots \qquad \text{and} \qquad abb\,\underbrace{aba}\,a\dots$$

which are overlapping. In the event communication service though, we are interested in separate, non-overlapping occurrences of the correlation. We therefore define the notion of a *trigger* on the sequence $\boldsymbol{s}$.

Definition 4. *A shortest match of an expression* x *on a sequence* $\boldsymbol{s}$ *of events is a* **trigger***, iff it does not overlap with a previous trigger on* $\boldsymbol{s}$*.*

Note that at the beginning of a sequence there can be no previous trigger, hence the definition is well founded. The second match in the above example therefore it is not a trigger. Intuitively, a state based acceptor resets at the point $abba \downarrow baa\dots$. We say a filter *triggers* whenever it completes a trigger.

2.4 The Parallel Combinator

There can be situations, in which all overlapping triggers are of interest to the receiver of a correlated event. Moreover though, as described in section 4, we want to be able to influence the behavior of a correlator without affecting ongoing computations of the filter. We hence introduce a parallel combinator `||`, which is exactly similar to the collection combinator `|` in its definition of a match and shortest match, but it allows overlapping matches:

Definition 5. *Let* $\mathbf{s}$ *be a sequence,* T_1 *be the set of triggers of expression* x_1 *on* $\mathbf{s}$*, and* T_2 *be the set of triggers of expression* x_2 *on* $\mathbf{s}$*. Then the set of triggers* T *for the expression* $x_1 \,||\, x_2$ *is the union* $T_1 \cup T_2$ *of the sets* T_1 *and* T_2*.*

Intuitively, the expressions x_1 and x_2 run independently from each other. Clearly it is not necessary to allow the use of the parallel combinator anywhere but in top level of the filter expression (this motivates the structure of the grammar in Fig. 1(a)).

2.5 The Result of the Filter Evaluation

Consider the expression $a|b$ which triggers whenever a or b appear in the input sequence. In case of triggering we want to know which one of the events, a or b actually arrived, since the result of the correlation delivered by the transformer (see section 3) might depend on it. Therefore we introduce the notion of an *active* subexpression.

Definition 6. *Let* $\mathbf{s}$ *be a sequence of events, and the subsequence* $\mathbf{s}'$ *of* $\mathbf{s}$ *be a trigger of expression* x *on* $\mathbf{s}$*. A subexpression* x' *of* x *is called* **active** *iff* $\mathbf{s}'$ *matches* x'*.*

E. g. b+c is an active subexpression of the expression a+$c|b$+c on the trigger ccb, while the subexpression a+c is not. For the trigger ac though, a+c is active, while b+c is not. Finally, for the trigger abc every subexpression of a+$c|b$+$c$is active.

To make the result of a subexpression accessible to the transformer, it has to be marked with a label.

Definition 7. *A label* l' *attached to a subexpression* x' *of an expression* x *is* **active** *iff* x' *is active.*

To illustrate we label two of the subexpressions of the previous example as $l_1 : (a$+$c) \,|\, b$+$l_2 : c$. On the trigger ca the label l_1 is active, on the trigger bc not. Note that l_2 is active on every trigger of the expression.

Upon completion of every trigger the filter propagates the set of active labels as result to the transformer. Note that the filter does not terminate after the first trigger, but it continues to trigger throughout the sequence of incoming events.

2.6 The Concrete Syntax of the Correlation

We define the correlators offered by the middleware communication channel in a separate input file called *correlation library*[2]. For simplicity we designed the

[2] See section 5.

form of the correlation definition closely similar to usual method or procedure definitions in imperative languages:

$$\begin{gathered}\textit{output-type}\ \texttt{correlation}\ \textit{name}\ \texttt{(}\textit{typed-identifier}_1,\ \textit{typed-identifier}_2,\ \ldots\texttt{)}\\ \textit{filter-expression}\ \texttt{\{}\ \textit{transformer}\ \texttt{\}}\end{gathered}$$

Here, the *output-type* is an event type defined in the CORBA IDL file[2] designating the type of the event sent out by this correlator. It is followed by the keyword `correlation` and an identifier *name* denominating the correlation. The events handled by this correlator are then provided as a list of typed identifiers which, in analogy to the parameters of a method, are later bound to event ports in the system assembly. The *filter-expression* defines the event subsequences upon which the correlation is supposed to trigger as an expression over the identifiers from the typed identifier list using combinators and labels in the syntax and semantics given previously in this section. Example:

```
Notification correlation AfterTimeout (TimeOut a, DataAvailable b)
                               a ; b {...}
```

Here, the identifier `a` will be bound to a port issuing `TimeOut`-events, while `b` is bound to a `DataAvailable` port. The correlation triggers on every sequence which contains a `TimeOut` and later a `DataAvailable`-event. The parenthesis "`{...}`" stands for the transformer, which is discussed in the next section.

3 The Transformer

3.1 Outline of the Transformer

The transformer provides the second step of our two phase model. As indicated in section 2.5, the input for the transformer is a set of active labels delivered by the filter. The function of the transformer is to assemble a new event based on various of the available incoming events and to push that event to the receiver(s) whenever a trigger is complete. In section 4 we will assign further objectives to the transformer, namely the possibility to perform dynamic changes to the correlator's behavior, to add more flexibility.

The transformer has two parts. The first part is an initialization, for which the discussion will also be deferred to section 4. The second part consists of case clauses branching on boolean expressions over the labels, assigning the values true to active and false to inactive labels respectively.

3.2 The Basic Transformer Syntax

An abbreviated grammar for the transformer is shown in Fig. 2. The substantial parts of the transformer are the `case`-clauses. Each case features a boolean expression over the labels. Upon a trigger, the body of *each* case, for which the associated label expression evaluates to *true* is executed.

```
     transformer ::= init ( case )*                          label-exp ::= disjunct ( | disjunct )*
            case ::= case label-exp : ( statement )*          disjunct ::= conjunct ( & conjunct )*
       statement ::= push event;                              conjunct ::= !conjunct
           event ::= identifier                                          | ( label-exp )
                     | new event-initial                                 | label
   event-initial ::= type { attr-assignments }                           | true
attr-assignments ::= ( attribute := identifier.attribute )*
```

Fig. 2. Basic Transformer Grammar.

For now there are three possible actions to take inside the body of a case clause: First, one of the events can be just propagated through, by giving the identifier of the event as argument to the `push` statement. A side condition is that the type of the event is a subtype of the declared output type of the correlation.Second, the correlator can assemble a new event of a given type, which again must be a subtype of the declared output type. The third, and at first sight trivial option is to do nothing at all. The section 3.4 though will show how this adds considerably to the possibilities of our approach.

3.3 The Transformer Output

For the assembly of an event we provide the `new` statement. It receives an event type defined in a separate file using the CORBA Interface Definition Language (IDL)[3] and a comma separated list of attribute assignments enclosed in curly brackets. For example the following lines define two event-types in IDL-syntax, where event `DataNotify` inherits from event `Notify`:

```
eventtype Notify {                    eventtype DataNotify : Notify {
   attribute short SourceID;             attribute float Value;
};                                    };
```

A transformer, given e. g. an input `DataNotify` event a, can assemble an output `Notify` event with the statement

```
push new Notify { SourceID = a.SourceID }
```

We make use of the inheritance subtyping given by the IDL event declarations, i. e. whenever e. g. `Notify` is defined to be the event type of an incoming event, the correlator accepts events of type `DataNotify` or any other subtype, similarly it can send out events of any type which is subtype of the declared output type.

3.4 Examples

Double Match. In [11], Boeing engineers discuss correlations in the context of the Boeing Bold Stroke framework. In their example, a receiving component is interested in notifications from three different event sources, referred to as A, B and C. A correlation occurs whenever either both, component A and B, or

[3] See section 5.

```
case l1: push new NavData
  { air = a.air, nav = b.nav }
case l2: push new NavData
  { air = a.air, nav = c.nav }
```

(a) Send data from both events

```
case l1: push new NavData
  { air = a.air, nav = b.nav }
case l2 & !l1: push new NavData
  { air = a.air, nav = c.nav }
```

(b) Take only b if present

```
case true: push new DataAvailable {}
```

(c) Send general notification

Fig. 3. Handling a Double Match Trigger.

both, component A and C have issued an event[4]. The filter expression describing this pattern is a+b|a+c, or equivalently a+(b|c). Consider the following stream of incoming events: ...cba. The sequence matches the expression, and hence is, assuming no previous overlapping trigger, a trigger for the filter. Note though, that the sequence is a match for both subexpressions a+b as well as a+c. Naturally, there must be a clear definition on how to handle this case. To discuss these, we label the subexpressions as follows: l_1:(a+b)|l_2:(a+c). *Fig.* 3 presents three different possibilities for the transformer to react to a trigger.

In Fig. 3(a) the transformer sends an event containing data from the incoming event a if l_1 is active, or an event containing data from event b if l_2 is active. If both labels are active though, as is the case with the above mentioned trigger, the transformer of Fig. 3(a) will generate *two* events. This complies with the policy informally described in [11].

Fig. 3(b) presents an alternative strategy, where the combination of event a with event b is favored over the combination of a and c. Again, this behavior is easy to specify by accessing the active labels: if l_1 is active the output is assembled from b. Only if l_1 is not active, the transformer uses c. A behavior like this, although it suggests itself in many common situations, is extremely complicated to describe in any of the previous approaches.

In many cases, a component may not be interested in particular payload values arriving in correlated events, i.e., the component simply needs to know that a correlation trigger has occurred. In this case, the transformer can be constructed to simply output an event with empty payload as shown in Fig. 3(c). Here, upon any trigger the same notification event is generated, regardless of the incoming events.

Interleaving Event. A primitive offered by other frameworks is the *non-interleaving* or *do-unless* correlation, expressed e.g. with `{e1;e2}!e3` in [7], or with **do**$\{\phi_1\}$**unless**$\{\phi_2\}$ in [10]. Common to these primitives is that some

[4] In one of the Bold Stroke examples, the *modal scenario*, this situation is given by two different steering queues, which correlate with a single air-frame as input to the combined display.

expression is pursued on the stream of incoming events until it is interrupted and reset by an interleaving event. For example, after the occurrence of `e1` in the above expression from the GEM framework, the correlator looks for `e2` or interrupts if `e3` comes in between. Similarly in the Stanford approach the expression ϕ_1 is evaluated in parallel with ϕ_2, and if ϕ_2 succeeds earlier, then the whole expression results in a *fail*.

Our model provides the same functionality without an additional primitive. Consider an expression x_0 which should not be interrupted by the completion of a second expression x_1. The filter expression $l_1 : x_0 \mid l_2 : x_1$ executes both expressions in parallel, resetting whenever either one triggers. As shown in Fig. 4(a) this is fully sufficient to prohibit interleaving of the two subexpressions, simply by ignoring the label l_2. Unlike previous approaches though, we can even safely handle a case where, similar to the double match example, both expressions complete with the arrival of the same event, e. g. by explicitly requiring the label l_2 not to be active when sending out the result. This is done by replacing the label expression `l1` by the expression `l1 & !l2` in Fig. 4(a). Further, whenever this expression is integrated as a subexpression into a larger context, it is easy to refer to either x_0 or x_1 by using the labels, e. g. to catch and handle the interleaving reset event, instead of ignoring it.

```
case l1: push new event
```

(a) Prohibit interleaving

```
case l1: push b
case l2: push a
```

(b) Most recent event

Fig. 4. Interleaving Semantics and Most Recent Event.

Most Recent. Consider a component interested in the accumulation of two events a and b. One possible filter which recognizes this accumulation is a+b. It is possible though to retrieve further information, e. g. about the order in which the events arrive. To achieve this, we expand and label the expression into an equivalent expression $l_1 : (a ; b) \mid l_2 : (b ; a)$. Still, this filter triggers whenever both, a and b are present in the incoming stream of events. The transformer body shown in Fig. 4(b) transfers the most recent of both events through to the subscriber.

4 Dynamic Changes

4.1 Changing Requirements

The communication structure in component based distributed systems is usually subject to dynamic modifications as it has to adapt to changes in the component's behavior as well as varying environmental properties. Especially the different operandi of the components referred to as *modes* cause frequent alterations to the communication. A correlator which is designed to work throughout the system's runtime has to offer possibilities to adjust both, filtering as well as transformation and propagation of events, to changing requirements.

In analogy to the modes of the components, some earlier approaches provide modes also for the correlator represented by mode guards, each of which in turn encapsulates a complete correlation definition. Nevertheless, according to our assessment of the component scenarios provided to us by the Boeing company, changes to the correlator rarely require an exchange of the whole definition by an entirely different one, instead the most common change is simply temporary absence of single events. Our approach aims to provide easy means to express this common scheme while still supporting more complicated dynamic adaption.

4.2 Cancelling Subexpressions

In our model, dynamic changes are realized by dropping or restoring subexpressions of the filter. We hence introduce two different states for each subexpression, *alive* and *aborted*. Similarly to the filter semantics given in section 2, we identify a label with the subexpression marked by the label, i. e. a label is aborted iff the marked expression is aborted and alive iff the marked expression is alive.

Definition 8. *Let x_a be an* **aborted** *expression. Then for any filter expression x_0 and any combinator $\oplus \in \{\texttt{;}, \texttt{|}, \texttt{+}, \texttt{||}\}$ we have that $\mathrm{L}(x_a \oplus x_0) = \mathrm{L}(x_0 \oplus x_a) = \mathrm{L}(x_0)$.*

In short, an aborted subexpression will simply be ignored by the filter. Note though, that for different combinators this definition has different implications, e. g. in an accumulation an aborted subexpression will be treated as "always present" while in a collection it will be treated as "never occurring".

Similarly to an aborted expression, any literal containing the attached label is also ignored in the label expressions guarding the cases in the transformer.

Definition 9. *Let l_a be an* **aborted** *label. Then for any label expression x_0 and any boolean operator $\oplus \in \{\texttt{\&}, \texttt{|}\}$ we have that $\lambda_a \oplus x_0 \equiv x_0 \oplus \lambda_a \equiv x_0$, where λ_a is either l_a or $!l_a$. An empty expression evaluates to* `false`.

Analogous to the filter expressions, this definition interprets an aborted label's literal as `true` in a conjunction and as `false` in a disjunction.

```
     init ::= ( commuter-stmt; )*        commuter-stmt ::= abort (label (, label)* )
statement ::= push event;                                 | revive (label (, label)* )
            | commuter-stmt;                              | toggle (label (, label)* )
```

Fig. 5. Transformer Grammar Extensions.

4.3 Additions to the Filter Syntax

Fig. 5 shows the extensions to the transformer grammar which enable the dynamic features of the correlator[5]. With the statements `abort` (l) and `revive` (l) label l and the subexpression marked by l can be dropped and restored, the `toggle` (l) statement switches between abort and alive state. By default every label is initially alive. Whenever a label is supposed to be in abort state initially, it has to be switched off in the initialization part of the transformer.

[5] Note the basic grammar in Fig. 2.

4.4 Simulation of a Mode-Based Approach

Consider a mode expression from [10] with n mode guard expressions $g_1, \dots g_n$ and enclosed correlator expressions $x_1, \dots x_n$:

$$\textbf{in}\ (g_1)\ \textbf{do}\ \{x_1\} \qquad \dots \qquad \textbf{in}\ (g_n)\ \textbf{do}\ \{x_n\}$$

We can easily simulate this functionality with the following filter expression:

$$m_1\texttt{:}\ x_1\ \texttt{|}\ \dots\texttt{|}\ m_n\texttt{:}\ x_n\ \texttt{||}\ l_1\texttt{:}\ g_1\ \texttt{|}\ \dots\texttt{|}\ l_n\texttt{:}\ g_n$$

and the following transformer

```
{ abort ($m_1$, $\dots m_n$);
  case $l_1$: abort ($m_1$, $\dots m_n$); revive ($m_1$);
  ...
  case $l_n$: abort ($m_1$, $\dots m_n$); revive ($m_n$);
  internal cases of the different expressions }.
```

Therefore the mode based approach offers no expressive power beyond our approach.

4.5 Example: Dropping an Unreliable Source

A common task for a correlator in Distributed Realtime Environment (DRE) applications is to accumulate all incoming events from e. g. a redundant sensor array and to send a combined event to the receiving component. We assume four sources referred to as A, B, C and D, with the events a, b, c and d. We further assume that a controlling component is able to determine the logical validity [9] of the data issued by the sources, and reacts to invalid data by issuing a shutdown event, telling the correlator to ignore the corresponding source. The filter expression hence is

```
ma:a + mb:b + mc:c + md:d || la:ca || lb:cb || lc:cc || ld:cd
```

Accordingly, the transformer is

```
{ case la: abort (ma);
  case lb: abort (mb);
  case lc: abort (mc);
  case ld: abort (md);
  case ma & mb & mc & md: push new DataAvailable {} }
```

If we want to enable the controlling component to be able to revive the dropped sources again, we can similarly use `toggle` instead of `abort`. Note though, that to cover every possible case with n redundant sensors a mode based approach as proposed by [10] would need up to 2^n different modes to achieve the same functionality. We are not aware of any support for dynamic modification of the correlator's behavior in previous works other than the above cited mode based proposal from Stanford.

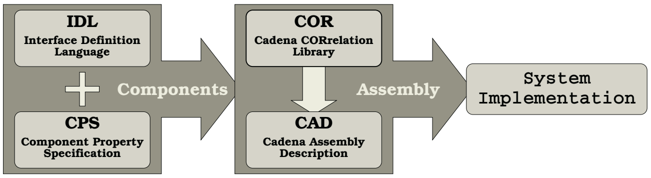

IDL
Interface Definition Language
CPS
Component Property Specification
Components
COR
Cadena CORrelation Library
CAD
Cadena Assembly Description
Assembly
System Implementation

Fig. 6. The Elements of a Cadena Specification.

5 Use in Cadena

We implemented the correlation framework described here in Cadena, a development environment we built for constructing distributed systems using the CORBA Component Model [4]. Cadena provides a wide range of support for modeling, analysis, and automatic code generation including facilities for specifying CCM component interfaces using CCM IDL, editors for allocating component instances and constructing connections between these instances, specifying component attributes, various forms of architectural-level analysis such as model slicing and model-checking against temporal specifications, and the ability to generate component code from IDL specifications using existing CCM implementations such as *OpenCCM*[8] (generating Java code) and *CIAO*[1] (generating C++ code). Cadena is implemented in IBM's *Eclipse*[3] open-source integrated development environment.

In Cadena, development begins by modeling components using CCM IDL which defines the external interfaces of components. Cadena provides an additional *component property specification* (`.cps`) file that is used to various lightweight semantic properties of a component including dependences between actions on a component's ports and an abstract transition semantics for the component (see Fig. 6).

Once components are modeled, a "system layout" is constructed in which instances of components are allocated and component ports of these instances are hooked together. Component instance connection information is held in *component assembly description* file. As Fig. 6 illustrates, correlations specified in `.cor` files can also be captured in the model. Various forms of static analysis and behavioral checking of the model work on internal representations formed from the component IDL, `.cps`, `.cad`, and `.cor` files.

Below the modeling level, component implementations are generated from CCM IDL compilers. Not only do these compilers generate the standard CORBA stubs and skeletons, they also generate a substantial amount of the code required to implement component infrastructure such as functionality for connecting ports, communicating events, and otherwise support interaction with a component's context.

In part of our larger project on development of Cadena infrastructure, other researchers at Kansas State have developed a flexible *Event Communication Framework (ECF)* [2] to augment the basic event propagation mechanisms of CCM and the lower CORBA layers. ECF is targeted towards optimizing communication between peers in a distributed system based on middleware in the presence of *event correlation* and *data replication*. Since ECF is independent of implementation (e.g., the underlying ORB or implementation language) and architecture (e.g., the topology of a particular application), it can be used seamlessly in any CCM-based application. CADENA processes correlator definitions in the `.cor` files along with connection information in the `.cad` file and synthesizes correlator implementation code to be linked into the CORBA communication layers and component container and server implementations. In this phase, as an optimization the framework may break a correlator into parts that realize subexpressions of the filter expression depending on the locality of components producing events occurring in the subexpressions. We refer to such subexpressions as *node local subexpressions*. This reduces network traffic by contributing internode network traffic only when a node local subexpression is satisfied. Likewise, as the mode of a component can affect the consumption/production of events by that component, by propogating this information upstream/downstream along the event flow path further optimization can be achieved. At present, ECF can use this information to dynamically configure the correlator, as described in Section 4, depending on the specified behavior and the usage context of the correlation. This can contribute to reduction in network traffic in a direct or cascading fashion.

Boeing's Open Experimental Platform (OEP) provides a set of scenarios which are representative of how DRE systems are built and used. With the above mentioned features, we have successfully realized all event correlations that occur in the scenarios in the above OEP. As the Boeing OEP is based on Boeing Bold Stroke avionics middleware (built on top of ACE/TAO real-time middleware), the handling of event correlations in a manner that is compatible with CCM specifications has removed one of the major obstacles in migrating the OEP into CCM from the non-standard component model which forms the basis of the concurrent Bold Stroke implementation and which lacks a substantial amount of infrastructure (e.g., deployment facilties and IDL code generation) that is provided by CCM.

6 Related Work

Our work was inspired from the early stages by previous work on event correlation from the GEM project [7] and a group from Stanford [10] also working on Boeing Bold Stroke. The Stanford model provides a rigorous formal background by defining the semantics of their definition language using automata similar to finite automata, called *correlation machines*. While this approach gives a convenient base for an implementation, we abstained from binding our semantics definition to a particular computational model, mainly because the implementation itself (part of our KSU colleagues work on ECF) continues to evolve as we work with middleware experts from the CIAO/ACE/TAO teams from Vanderbilt and Washington Universities to adapt the standard CCM event infrastructure into a form that is better suited for real-time applications. In particular, this effort is investigating, e.g., computation time and network traffic optimizations using partial pre-evaluation of correlations close to the event source.

The Reflex framework [6] provides a fully implemented correlation engine, together with a specification language called *Policy Definition Language*. Reflex generates C++ correlators intended to monitor network events in an internet like structure. Accessible semantic definitions are informal though. Note, that CCM uses the notion of events primarily for communication, while Reflex and other work concentrates on events as a means to monitor system behavior by some kind of controller.

7 Conclusion

We have presented a flexible framework for event correlation based on the two-phased approach of event filters and event transformers. Our directions within the work have been heavily influenced by the special needs of complex high-reliability real-time systems, and our desire to be able to develop such systems using the CCM framework. While previous approaches define the complete correlation in a single expression, we believe that employing our two phase model eases the assembly of meaningful output significantly.

Not only does the separate transformer stage make it easier for the framework to fit into the CCM type system, it also lends itself to a variety of interesting capabilities required for the real-time domain. For example, in collaboration with other middleware researchers, we are investigating approaches for attaching various real-time and quality of service properties to events. Given these extensions, transformers are also used to transform priorities, expiration times, and to implement drop strategies for events. For further development of our model, policies about event overwriting are being investigated, e. g. for events carrying sensory data the most recent event has priority over previous, while for an error message the earliest occurrence is more important. In addition, we are investigating how such policies may be captured in an extended event type system.

References

1. Component-integrated ACE ORB, a C++ implementation of CCM. Available at http://www.cse.wustl.edu/~nanbor/projects/CIAO/.
2. Event communication framework. Available at http://neo.projects.cis.ksu.edu.
3. Eclipse, an open extensible ide and tool platform written in java. Available at http://www.eclipse.org.
4. J. Hatcliff, W. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 2003 International Conference on Software Engineering (ICSE'03)*, May 2003.
5. G. Jung, J. Hatcliff, and V. P. Ranganath. A correlation framework for the CORBA component model. Technical Report 03-9, Kansas State University, Department of Computing and Information Sciences, 2003.

6. S. Louvau, D. Chen, S. Jackson, P. Devanbu, and M. Gertz. Reflex – the customizable event correlation system. http://reflex.cs.ucdavis.edu/.
7. M. Mansouri-Samani and M. Sloman. Gem: A generalised event monitoring language for distributed systems, 1997.
8. OpenCCM, a Java implementation of CCM. Available at http://openccm.objectweb.org.
9. R. J. Richards, G. W. Daugherty, D. A. Haverkamp, and C. B. Netto. Middleware-based automatic source selection. Rockwell Collins internal, September 2002.
10. C. Sánches, S. Sankaranarayanan, H. Sipma, T. Zhang, D. Dil, and Z. Manna. Event correlation: Language and semantics. In *Proceedings of EMSOFT'03*, volume 2855, pages 323 – 339. ACM, Springer, September 2003.
11. D. Sharp. Challenge problems for the model-based integration of embedded software weapon system open experimental platform. Part of the Boeing OEP software., July 2001.
12. H. Sipma. Event correlation: A formal approach. Technical Report Draft, Stanford University, July 2002.

Cadena: An Integrated Development Environment for Analysis, Synthesis, and Verification of Component-Based Systems[⋆]

Adam Childs, Jesse Greenwald, Venkatesh Prasad Ranganath,
Xianghua Deng, Matthew Dwyer, John Hatcliff, Georg Jung,
Prashant Shanti, and Gurdip Singh

Department of Computing and Information Sciences
Manhattan, KS 66506, USA
`http://cadena.projects.cis.ksu.edu`

Abstract. This tool paper gives an overview of Cadena – an integrated environment for building and modeling systems built using the CORBA Component Model (CCM). Cadena provides facilities for defining component types using CCM IDL, specifying dependency information and transition system semantics for these types, assembling systems from CCM components, visualizing various dependence relationships between components, specifying and verifying correctness properties of models of CCM systems derived from CCM IDL, component assembly information, and Cadena specifications, and producing CORBA stubs and skeletons implemented in Java. Cadena has been applied to build applications in Boeing's Bold Stroke framework for avionics mission-control systems. Cadena is implemented in IBM's Eclipse open-source IDE and is freely available.

As software systems become more distributed, developers are increasingly turning to component-based development frameworks such Java Enterprise Beans (EJB) and the CORBA Component Model (CCM) to manage the complexities associated with building distributed systems. These frameworks aid application developers by providing services for common aspects such as distributed deployment, event notification, transactions, persistence, and security. Moreover, they use accepted design patterns (e.g., the event-oriented observer pattern) which enables a significant amount of code to be auto-generated. Component-based frameworks are also attractive because the relatively loose coupling between components facilitates reuse and allows systems to evolve gracefully as old components are switched out for new ones.

Even in the domain of distributed real-time embedded (DRE) systems where hard/soft deadlines and minimal foot-print requirements traditionally have led

⋆ This work was supported in part by the U.S. Army Research Office (DAAD190110564), by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), by Lockheed-Martin, Rockwell-Collins and by Intel Corporation (Grant 11462).

developers to eschew sophisticated middleware solutions, component-based infrastructures are growing more popular because hardware advances allow real-time and embedded requirements to be more easily achieved. In addition, component-based infrastructures provide a framework for systematically introducing important domain aspects such time-triggered notification, real-time scheduling, and fault tolerance.

There is a wide body of literature dealing with the theory of modeling distributed systems and automated analysis of high-level state-based models using state-space exploration techniques such as model-checking. However, despite the popularity of component-based frameworks and their potential to be utilized in mission- and safety-critical applications, relatively little has been done to scale up these analysis techniques for the purpose of providing automated analysis tools for component frameworks. This is particularly the case with CCM – partly due to the fact that the CCM specification as part of CORBA 3.0 has only recently been finalized. Popular tools such as Rational Rose do not even provide design support for CCM yet.

To investigate the effectiveness of a variety of behavioral analysis techniques for component-based systems, we have built *Cadena*[1] – an integrated development environment for high-assurance CCM-based systems.

Cadena provides the following fully implemented capabilities for development of CCM systems.

– A collection of light-weight specification forms that can be attached to IDL to specify mode variable domains, intra-component dependencies, and component state-transition semantics. These forms have a natural refinement order so that useful feedback can be obtained with little annotation effort, and increasing the precision of annotation yields more precise analysis. In addition, Cadena specifications allow developers to specify the same information in different ways, achieving a form of *checkable redundancy* that is useful for exposing design flaws.
– Dependency analysis capabilities that allow tracing inter/intra-component event and data dependencies, as well as algorithms for synthesizing dependency-based real-time and distribution aspect information.
– A novel model-checking infrastructure (based on our Bogor model-checking framework [4]) dedicated to event-based inter-component communication via real-time middleware enables system design models (derived from CCM IDL, component-assembly descriptions and annotations) to be model-checked for global system properties.
– Java component stub and skeleton code generated using the OpenCCM [2] CCM IDL to Java compiler.
– A component assembly framework supporting a variety of visualization and programming tools for developing component connections.

[1] "Cadena" is a Spanish word meaning "chain" or "network". Cadena is also an acronym for Component Architecture Development ENvironment for Avionics systems.

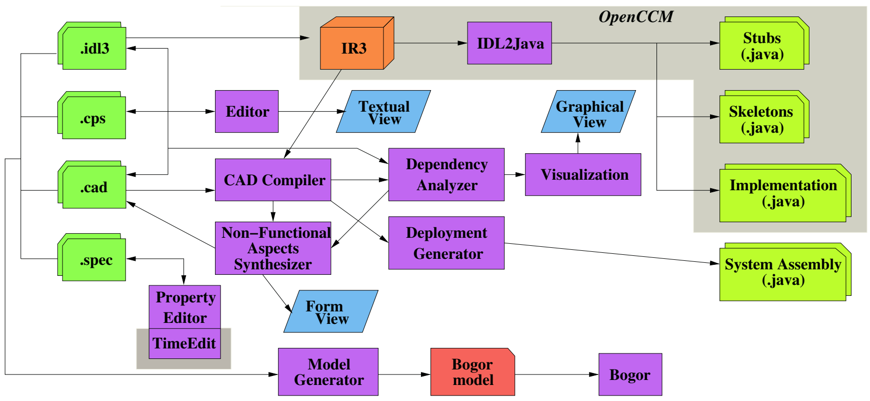


Fig. 1. Cadena architecture.

– A CCM deployment facility dedicated to the Boeing Bold Stroke architecture (static component connections with a real-time event-channel) that allows deployment code to be automatically generated.
– The Cadena tools are implemented as plug-ins to IBM's Eclipse IDE. This provides an end-to-end integrated development environment for CCM-based Java systems.

Figure 1 displays the internal structure of the Cadena toolset. Cadena projects contain four high-level specification forms: a CORBA 3 IDL file that defines the structure of component types, a Cadena Property Specification (CPS) file that specifies various aspects of component behavior including abstract state transition semantics and information that specifies dependences between component ports, a Cadena Assembly Description (CAD) that specifies the components instances that form the system, the connections between them, along with other real-time and distribution property information, and a specification file that stores information about the desired correctness properties of the system. These input artifacts are created using customized editors built using Eclipse plug-in facilities. In particular, the CAD format has a textual editor, a graphical editor, and a form-based editor that allows one to easily define different projections of the component assembly (e.g., connections only, particular component attributes only, etc.). The graph structure described by the CAD is the basic data structure that is used by the dependency analyzer, the graphical view displayer, and the deployment code generator (which generates Java code to allocate and connect components).

Cadena uses the OpenCCM tools [2] to generate system implementations. The OMG CORBA 3.0 specification standardizes a strategy for compiling IDL (of which the CCM IDL is part) down to CORBA IDL 2, which can then be translated to an underlying implementation language such as Java or C++. This translation process automatically generates a substantial amount of infrastructure code for tasks such as component creation and connecting and disconnecting ports. The output code contains the usual CORBA *stubs* and *skeletons*, along with skeleton *implementations* of component methods and event handlers. With this code generation, the developer only needs to implement event handlers and methods on provided interfaces. In future work we are exploring the extension of CCM-based code generation strategies to integrate code generation for component handler state-machines and global synchronization policies.

When building systems with Cadena, we intend for developers to take the following steps: (1) load a library of domain-specific components and associated CPS specifications, (2) define new project-specific components and associated behavioral CPS specifications, (3) use CAD editors to configure connections between components, (4) use dependency viewer to examine dependencies, (5) use non-functional aspect synthesis tools to attach distribution and rate information, (6) specify desired global correctness properties, (7) generate a transition system model and model-check correctness properties, and (8) revise system based on feedback from analysis tools.

Up to this point, Cadena has been applied primarily to develop representative applications from Boeing's Bold Stroke avionics mission control software framwork. We have worked with engineers from both Boeing and Rockwell-Collins with a goal to design Cadena and its associated use methodology so that it could be integrated into the actual Bold Stroke development process. In fact, we believe that this "customer-driven" context is one of the things that makes this work interesting and relevant: we address analysis of widely-used general purpose middleware frameworks and languages, and we design the functionality and features of our analysis tools to mesh with an actual industrial development process.

Although Cadena was originally targeted to the avionics domain, it is useful in many respects for CCM development in general. Even though it currently emphasizes Java in its back-end facilities, since CCM is language-neutral, Cadena's front-end design capabilities are not Java dependent. For example, we are also working closely with researchers developing CIAO [1] (a C++ CCM implementation based on the ACE/TAO real-time middleware framework) to integrate CIAO into Cadena, and to refine the Cadena APIs to support specification and modeling of real-time and quality-of-service properties.

There are other development systems that support several important aspects for DRE systems that Cadena does not, such as timing and schedulability analysis, reliability and fault analysis, as well as sophisticated deployment strategies. The primary motivation for our work is to build a system that is robust enough for development of real systems with the goal of assessing the effectiveness of applying static analysis, model-checking, and other light-weight formal methods to CCM-based systems.

The technical foundations of Cadena were presented in [3]. More information about Cadena including a public distribution, papers, tutorials, talks, example repository, and guidelines for other researchers wishing to integrate their analysis tools or CCM implementations into Cadena can be found at the Cadena web site [5].

References

1. DOC Group – CIAO Development Team. CIAO Website. `http://www.cs.wustl.edu/~nanbor/projects/CIAO`, 2003.
2. GOAL. The OpenCCM platform. `http://corbaweb.lifl.fr/OpenCCM/`, 2002.
3. J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 25th International Conference on Software Engineering (to appear)*, 2003.
4. Robby, M. B. Dwyer, and J. Hatcliff. Bogor Website. `http://bogor.projects.cis.ksu.edu`, 2003.
5. SAnToS Laboratory. Cadena Website. `http://cadena.projects.cis.ksu.edu`, 2003.

Actor-Centric Modeling of User Rights

Ruth Breu[1] and Gerhard Popp[2]

[1] Research Group "Quality Engineering", Universität Innsbruck
Institut für Informatik, A-6020 Innsbruck, Austria
Ruth.Breu@uibk.ac.at

[2] Software & Systems Engineering, Technische Universität München
Institut für Informatik, D-85748 Garching b. München, Germany
Gerhard.Popp@in.tum.de

Abstract. In this paper we present a novel approach for the predicative specification of user rights in the context of an object oriented use case driven development process. We extend the specification of methods by a permission section describing the right of some actor to call the method of an object. Moreover, we introduce a representation function that describes how actors are represented internally in the system. As syntactic and semantic framework we use a first-order logic with a built-in notion of objects and classes provided with an algebraic semantics. We demonstrate that our approach can be realised in OCL.

1 Introduction

The requirement of protecting data from unauthorised user access is as old as multi-user computing. Applications such as ERP systems or health information systems with hundreds or thousands of users handling sensible data offer sophisticated mechanisms for rights modeling. With the new web technologies the importance of data protection mechanisms will even grow. The more companies will open their core business processes to external partners the more important the enforcement of access rights will become.

Data protection is intimately connected with two aspects – authentication on the one hand side and access control on the other side. Authentication aims at identifying actors (persons or external systems) interacting with the system. Access control is concerned with the protection of information resources.

With the prominent RBAC model an adequate paradigm for implementing access rights has been developed [1,2,3]. Access rights in this model do not adhere to single users but are associated with *roles*. Basically each user may have one or several roles and each role is associated with a set of *permissions*, where each permission defines the kind of access (the operation, e.g. read, update) to some object. Today the RBAC model is one of the most established access models. This model is supported by many systems like operation systems, data bases and middleware platforms.

Despite of the role paradigm, data access control remains a complex task in real applications. In particular, data access control in most cases concerns different layers ranging from the user interface and the application layer to the database. Moreover, upcoming inter-organisational applications require novel (credential-based) techniques for enforcing user rights [4].

Despite the complexity of the task, an aspect neglected so far in the literature has been the analysis and design phase of user access models. Developing user right concepts for applications in areas like health care, e-government or knowledge management requires both an implementation-independent analysis framework and a step-by-step approach. Moreover, since user access models have to be developed in close cooperation of system designers and clients user rights modeling has to be integrated into the requirements engineering process.

In this paper we present an approach to the specification of user access rights satisfying these needs. In particular, our approach is based on the following three basic ideas.

First, we conceive user rights modeling as a task within the context of an object oriented development method such as the Unified Process [5] or the V-Model [6]. This means that our method is entirely integrated within the context of business process modeling, use case modeling and systems analysis. More generally, the approach we present in this paper is part of a process model for security engineering [7]. This process model extends an object oriented kernel model by techniques, artifacts and activities supporting the systematic construction of security-critical systems.

Second, our method supports the stepwise development of user right models. This ranges from informal textual statements to a complete predicative specification. The implementation independent specification of user rights has several advantages. The most important one is that the model developed is a compact and concise representation of knowledge involving many parts of the implementation. Moreover, complete user right models have the potential to be automatically transformed into code.

Third, our approach is provided with a formal semantics in an algebraic setting. On the syntactic side we extend the specification of each method by a *permission section* describing for each actor (or role) if this actor has the right to call this method on an object of the given class. Since actors play a central role for the specification of access permissions we call our method *actor-centric*. The permission is described by a first-order predicate over a structure with a built-in notion of objects or, in the context of UML, by an OCL statement [8,9].

Additionally, we introduce a representation function to describe how actors are represented internally in the system. This representation function is an abstraction of the authentication procedure in the implementation and allows specifications of the kind *"the user has the right to view his/her own data"*. The semantics of permissions and of the representation function can be embedded in a straight forward way in the algebraic theory presented in [10,11].

Related work in the literature mainly deals with the RBAC approach (e.g. [1,2,3]). Our approach goes beyond in several respects. First, we are concerned with the development process of user right models in the context of object oriented modeling techniques. Moreover, we provide an increased expressiveness by supporting arbitrary first-order predicates to specify user rights.

An approach with similar expressiveness is [12]. While this approach has been designed with the primary goal of code generation, our focus has been the development of concepts adequate for the whole development process. A scheme of user rights modeling in the context of use cases has been first presented by [13]. We overtake some of these ideas, but present a more elaborate theory. Further references in a specific setting are [14,15] dealing with the development process and checking of user rights in SAP applications.

The rest of this paper is organised as follows. In section 2 we give some background information. Section 2.1 gives an overview of the artifacts of a kernel object oriented method our approach is based on. Section 2.2 sketches the syntactical and semantical specification framework. A formal model of user rights is given in section 3, which is divided into a description of the representation function (see section 3.1) and permissions (section 3.2). The specification of permissions in OCL is presented in section 4 and extensions are introduced in section 5. In section 6 a conclusion is drawn.

In the sequel we assume the reader to be familiar with basic concepts and notations of object oriented modeling with UML and OCL.

2 Basics

In this section we first give a short overview of the modeling context our approach is based on. Throughout this paper we will use as running example a case study based on *TimeTool*, a software project that was realised in our research group. *TimeTool* is a portion of a project management tool allowing team workers to account worked hours for projects and allowing project managers to supervise project budgets.

2.1 The Core Object Oriented Artifacts

In this section we will shortly characterise the core artifacts of the object oriented process together with the dependencies.

The Business Model captures the organizational environment of the IT-system. The model describes

- Who (or, more precisely, what roles) act in the business domain (*actors*)
- What *activities* the actors perform
- Which *objects* the activities need as input and which objects they produce as output

In *TimeTool* the actors are the *project manager*, the *team worker* and the *administrator*. Example activities are *Account Worked Hours* and *Post Adjustment* (performed by the team worker or the project manager) and *Prepare Monthly Report* (performed by the project manager). Example classes in the application domain are the *project*, the *accounting* and the *team worker.*

Actors, activities and objects are modeled by several diagrams. In the UML context this comprises *activity diagrams* modeling business processes and activities, and *class diagrams* modeling the organizational structure of the company and the structure of static concepts (users, projects, accountings, etc.).

The System Requirements describe a black box view of the system to be developed based on the concept of *use cases*. Each use case corresponds to a coherent interaction between some actor and the system. The use cases together describe the whole functionality of the system. Basic concepts of the use case model are

- the *actors* interacting with the system
- the *use cases*
- the *objects* being involved in the use cases

In extension to business modeling the actors in the use case model might not only represent human beings (like the *team worker* and the *project manager*) but also external systems. For instance, additional actors in *TimeTool* are the web browser and the human resources system which hosts the databases with available staff and to which the system connects to for data import. Sample use cases are *Account Worked Hours*, *Adjustment Posting* and *View Statistics*.

The System Requirements basically consist of two descriptions, the *use case diagram* together with the *textual description* of use cases, and the *class model* describing the static concepts.

Commonly, the class model of the System Requirements is the same or a refined version of the class model of the Business Model. Figure 1 depicts the class model for *TimeTool*.

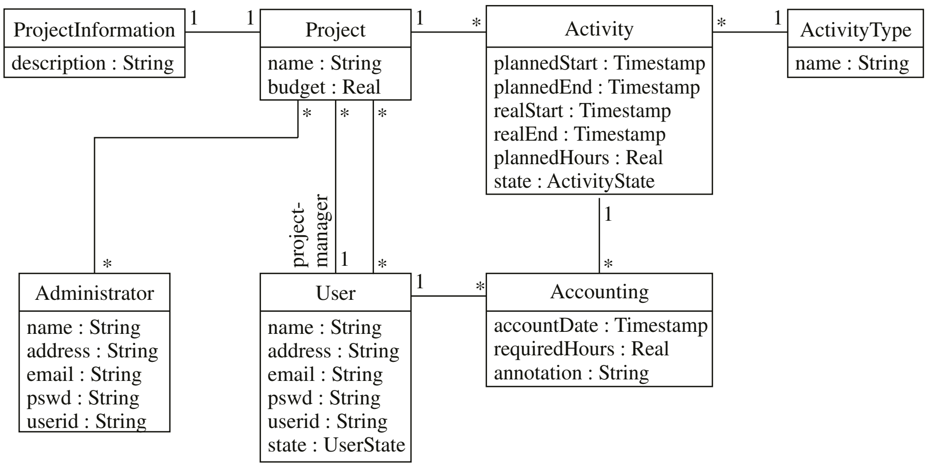


Fig. 1. Class Diagram of the *TimeTool* Example

The Application Architecture refines the level of description. More precisely, the system is divided into a set of *logical components*. Each component is responsible for a portion of the system structure and behaviour. Interfaces enable the independent development of the system components.

Concerning the design of the system behavior the textual descriptions of the use cases are refined into *scenarios* in the Application Architecture. The scenarios describe the use cases as message flows between objects. That way, an object oriented view of the whole system is achieved. In the UML context the Application Architecture typically consists of the following diagram types: One or several *component diagrams* containing components, interfaces and the related classes, a set of *sequence diagrams* each one related to some use case, and *state diagrams* modeling complex processes or class interfaces.

Business Model, System Requirements and the Application Architecture have in common that they rely on an application oriented system view and are independent of any technical platform.

The Software Architecture is based on an implementation oriented view of the system. In this model the hardware and software platform is chosen and roughly described. Since we will not deal with the platform dependent level in this paper we do not go into more detail at this place.

2.2 The Specification Framework

As specification framework we use the specification language P-MOS [10,11]. P-MOS supports first-order predicates with a built-in notion of objects and is provided with a semantics in an algebraic setting. The kernel syntactic constructs can be found below.

P-MOS can be compared in its expressiveness with OCL but provides the full flexibility of an algebraic specification language. In our approach P-MOS serves as an intermediate language for developing concepts and providing a semantics. As we will demonstrate OCL can be used as target language within our method.

P-MOS Expressions. Each P-MOS expression is based on a class diagram. The expression describes a navigation in an object structure delivering some result.

Semantically each P-MOS expression is interpreted in the context of a so-called object environment describing a concrete object structure over the class diagram given. P-MOS is a hybrid language which means that we distinguish between basic types and class types. While an expression of some basic type (such as *Bool* or *int*) denotes a value (*true*, *false*, *0*, *1*, ...) an expression of a class type denotes a reference to an object in the given object structure.

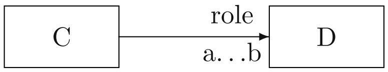
P-MOS expressions are built by the application of one of the six rules that can be found below. We assume to be given a set of basic data types (such as *Bool* and *int*) and type constructors. In particular, we assume a type constructor *Set[_]* describing sets of arbitrary elements.

(1) Basic Functions. Let *f*: $(s_1, \ldots, s_n)$ *s* be a function in a basic data type (such as *+: (int, int) int* or *true: () Bool*). If $e_1, \ldots, e_n$ are P-MOS expressions of type $s_1, \ldots, s_n$ then $f(e_1, \ldots, e_n)$ is a P-MOS expression of type *s*.

(2) Variables. Let X be a set of typed variables. Then each variable x of type s is a P-MOS expression of type *s*.

(3) Attributes. Let *A:T* be some attribute of class C (*T* either being some class name or some basic data type) and *e* be some P-MOS expression of type *C*. Then *e.A* is a P-MOS expression of Type *T*.

(4) Assocations. Let us assume an assocation



in the class diagram. Then *e.role* is a P-MOS expression of type *Set[D]* (*D*, resp. in the special case a...b = 1...1) if *e* is expression of type *C*. If the role name is missing then the navigation expression is constructed by *e.d* (the class name *D* written in lower case).

(5) Generalisation. If *e* is P-MOS expression of type *C* and *C* is subclass of class *D* (in the transitive closure) then *e* is P-MOS expression of type *D*.

(6) State-Based Functions. Let ***funct*** *f:*$(T_1,\ldots,T_n)$ *T* be a state based function (where $T_1,\ldots,T_n$, *T* are either basic types or classes) and $e_1, \ldots, e_n$ are P-MOS expressions of type $T_1,\ldots,T_n$. Then $f(e_1, \ldots, e_n)$ is a P-MOS expression of type *T*.

A state-based function describes a generic navigation in the given object structure based on *n* parameters and delivering a result of type *T* (either a basic value or some object reference). The properties of a state-based function are defined by P-MOS predicates as defined below.

Query operations in the class diagram include state-based functions in the following way:

For each query operation $f{:}(x_1{:}T_1,\ldots,x_n{:}T_n){:}T$ in class C we define a state based function *funct* $f{:}(C,T_1,\ldots,T_n)\ T$ and write expressions $f(e,e_1, \ldots, e_n)$ as $e.f(e_1, \ldots, e_n)$ in the usual way.

P-MOS Predicates. Based on the notion of P-MOS expressions P-MOS predicates are formed in the usual way as given in table 1.

Table 1. P-MOS Predicates

P-MOS Predicate	Element Description
e1 = e2	*e1*, *e2* P-MOS expressions of the same type
$\neg P$, $P1 \vee P2$, $P1 \wedge P2$, $P1 \Rightarrow P2$, $P1 \Leftrightarrow P2$	*P*, *P1*, *P2* predicates
$\forall x{:}T.P$, $\exists x{:}T.P$	*P* predicate, *x* variable, *T* type expression

Table 2. Access Rights from Team Workers and Project Managers

actor → ↓ *class*		**Team Worker**		**Project Manager**
Accounting	R:	all own accountings (independent of the project)	R/W/C:	all accountings of activities of own projects (i.e. where actor is the project manager of)
	W/C:	own accountings from released activities		
Activity	R:	all activities from projects allocated to the actor	R/W:	all activities of own projects
			C:	–
	W/C:	–		
Project	R:	all projects	R:	all projects
	W/C:	–	W/C:	–
User	R:	all users	R:	all users
	W/C:	–	W/C:	–

3 Formal Modeling of User Rights

The central notion for capturing individuals and their roles in business process modeling and use case modeling is that of an *actor*. For instance, in the business process model an actor stands for the person (or, more precisely, for the role of this person) being involved in the business process. In extension, actors in the use case modeling also may stand for the roles external systems play.

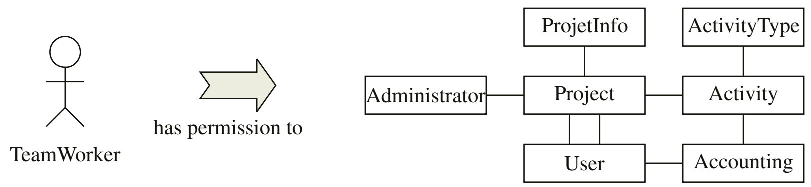


Fig. 2. Actors have Permissions on Objects.

The key idea to the modeling of user rights in our approach is that actors have some kind of permissions with respect to objects of the class model (see Figure 2). In this respect our user right model both refers to the model containing the actor (business process model or use case model) and to the class model. The separation of role concept and classes has the advantage that the way how roles are represented in the system has not to be fixed within requirements elicitation.

In early phases of the development we may wish to specify user rights in an informal, textual way. Table 2 depicts such a textual user rights model for our case study. The informal model contains coarse-grained permission categories

(R = Read, W = Write, C = Create) for each class. The textual statements characterise the objects of the given class which the actor may read, write or create. For large parts of an application such a coarse-grain description may be sufficient. For critical parts we require more fine-grained ways to express user rights. For that reason we offer a specification mechanism at the level of methods. More precisely, each *method* m in *class* C is associated with a *permission*

$$perm_{C,m}$$

specifying under which condition an actor has access to call the method on an object of the given class. In section 5 we will present a mechanism to aggregate permissions supporting a more coarse-grained level of detail.

What is missing in the framework sketched so far is a connection between actors and classes. Such a connection is required in cases where permissions refer to the actor himself like in the example the *team worker has read permissions to own accountings.* In order to support such kinds of specifications we provide a function

$$rolerep$$

mapping actors to objects of some class. This class (in most cases some class like *User*) is the internal representation of actors. In fact the representation function is an abstraction of the authentication procedure in the implementation. More information on that will be given in section 3.1.

To conclude this section we shortly summarise in table 3 the development steps of a user rights model in the context of the object oriented process.

Table 3. Development Steps of a User Rights Model

Development Step	**Activities**
Business Process Model	Informal description of actor permissions in tables.
Use Case Model	Adaptation of the informal model to the actors of the use case model (e.g. including extended systems). If possible development of a first formal model.
Application Architecture	Development of a complete formal model.

In the context of iterative development the abstract user right model has to be adapted and extended in each iteration (e.g. concerning new classes).

In the sequel we will present the formal mechanism of method permissions (section 3.1) followed by the representation function (section 3.2).

3.1 The Function *rolerep*

As motivated in the preceding section the function *rolerep* maps actors to their internal representation. In order to provide a homogeneous specification framework within P-MOS we internally extend the class diagram by a class hierarchy representing the actor roles.

In particular, we introduce a superclass *ACActor* modeling all kinds of actors. Subclasses of the class *ACActor* are all actors defined in the business process or use case model, respectively. Actor hierarchies in the use case model are transformed in a corresponding class hierarchy. In the example, we obtain subclasses *ACAdministrator*, *ACTeamWorker* and *ACProjectManager*, where *ACProject-Manager* is a subclass of *ACAdministrator* (see Figure 3). In our model actors

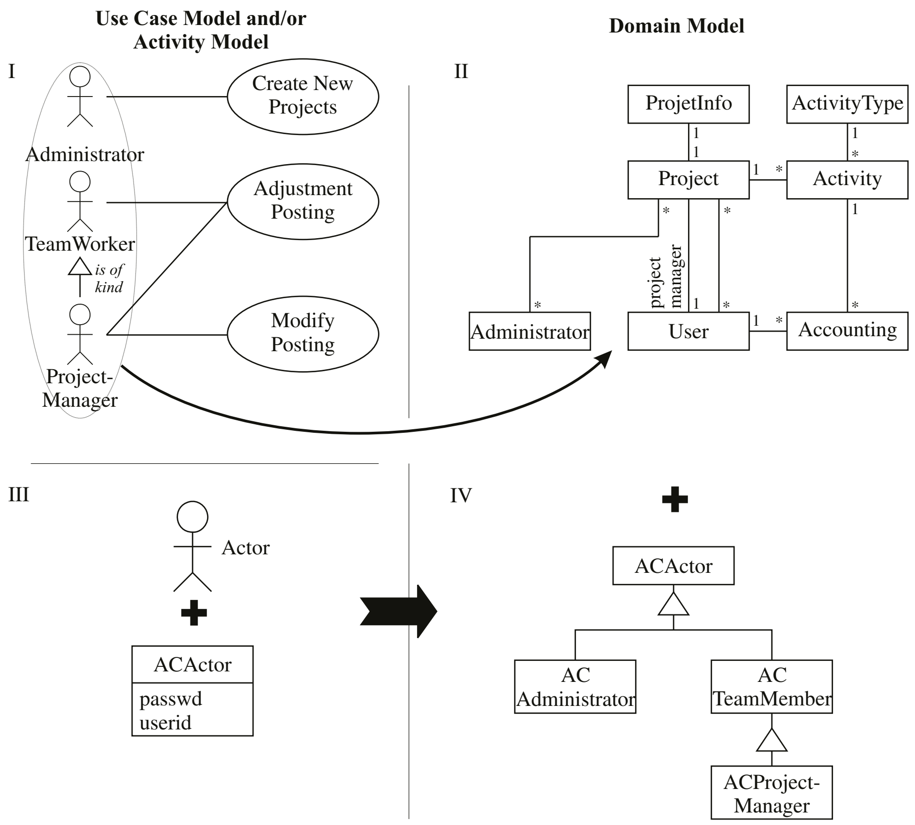


Fig. 3. The Extension of Actor Classes for an Actor–User Mapping.

also may have attributes. These attributes represent the input that is required to authenticate the actor (human being or external system) within the system. In most cases the attributes are the userid and the password, but also biometric data, credentials or "no information" (if the actor is anonymous to the system) are possible.

Formally, the (state-based) representation function *rolerep* has the functionality

$$\textbf{\textit{funct}}\ rolerep : (ACActor)\ Object$$

where *Object* is the superclass of all classes (like in Java).

The function *rolerep* is part of the actor class $ACActor$ and can be specified in this class for all actor types, or in each subclass for specific actor types. Equivalently the function *rolerep* can be conceived as query operation of functionality $rolerep : ()\, Object$.

As example we present the specification of the *rolerep* function for the two actors *TeamWorker* and *Administrator* of the *TimeTool* example.

ACTeamWorker

passwd: String
userid: String

rolerep : () Object

$$\begin{aligned} &\forall tw : ACTeamWorker \,.\, \forall u : User \,.\, tw.rolerep() = u \;\Rightarrow \\ &\qquad u.state = \sharp active \,\wedge \\ &\qquad tw.userid = u.userid \;\wedge\; tw.passwd = u.passwd \end{aligned}$$

ACProjectManager

rolerep : () Object

$$\begin{aligned} &\forall pm : ACProjectManager \,.\, \forall u : User \,.\, pm.rolerep() = u \;\Rightarrow \\ &\qquad \exists p : Project \,.\, p.projectmanager = u \end{aligned}$$

The specification of actors including the representation function in our modeling framework is part of the actor description in the use case model. During construction the representation function is implemented by the authentification procedure.

3.2 Permissions

Permissions are method preconditions associated with the semantics that the corresponding method can only be executed if the permission expression is evaluated to true at the beginning of the execution. Since the basis for our approach is the *fail-safe defaults principle*, every method execution which is not permitted explicitly through a permission is prohibited. Each method permission may depend on the calling actor, on the actual object, and on the actual parameters of the method call. Thus, permissions are state-based functions of the kind

$$\textbf{\textit{funct}}\; perm_{\,C,m} : (ACActor, C, T_1, \ldots, T_n)\, Bool$$

where C is a class, and m a method in C of the form *m-id*: $(x_1 : T_1, \ldots, x_n : T_n)\, T$ (In the special case of create methods or class methods the parameter of type C is omitted).

The properties of method permissions are specified by P-MOS predicates describing conditions over the given object structure.

In the following we will give a few examples of permission specifications. We model some access rights from Table 2 for the *team worker* and the *project manager* of our *TimeTool* example.

Example 1a: As first example we specify the permission that a *team worker* can read all his own accountings, independent of the project the accounting belongs to. As sample we use the method *getAccountingDate()* as representative for a reading method.

$$\begin{aligned}&\forall tw : ACTeamWorker \;.\; \forall a : Accounting \;.\\ &a.user = tw.rolerep\,()\\ &\quad \Rightarrow\; perm_{\,Accounting,\, getAccountingDate}(tw, a)\end{aligned}$$

The expression states, that the user object, that is linked with the accounting object must be the internal representation of the given *team worker*. Only if this expression evaluates to *true*, the *team worker* has access to the *getAccounting-Date()* method of the class *Accounting*.

Example 1b: As second example we consider the permission that a *project manager* can read all accountings associated with his own projects. Again we specify the permission of *getAccountingDate*.

$$\begin{aligned}&\forall pm : ACProjectManager \;.\; \forall a : Accounting \;.\\ &a.activity.project.projectmanager = pm.rolerep\,()\\ &\quad \Rightarrow\; perm_{\,Accounting,\, getAccountingDate}(pm, a)\end{aligned}$$

Example 2: A further permission for the *team worker* is, that he can only write accountings to released activities. Activities are associated with a state which may be set to *released* or *frozen*. In this way, it can be prohibited that somebody manipulates accounting objects related to finished activities. The permission is given in the following predicate:

$$\begin{aligned}&\forall tw : ACTeamWorker \;.\; \forall a : Accounting \;.\\ &a.user = tw.rolerep\,() \;\wedge\\ &a.activity.state = \sharp released\\ &\quad \Rightarrow\; perm_{\,Accounting,\, writeAccountingDate}(u, a)\end{aligned}$$

Example 3: As last example we study the permission of a create method. The *project manager* can only create accountings from activities of own projects. Here we assume for short, that the create method has the activity and the user the accounting refers to as only attributes.

$$\begin{aligned}&\forall pm : ACProjectManager \;.\; \forall ac : Activity \;.\; \forall u : User \;.\\ &ac.project.projectmanager = pm.rolerep\,()\\ &\quad \Rightarrow\; perm_{\,Accounting,\, create}(pm, ac, ac)\end{aligned}$$

Permission Inheritance. There are two aspects where the specification of permissions interferes with the concept of ineritance.

The first aspect is related with the hierarchy of actors (see quarter IV of Figure 3). In our example the *project manager* is a special kind of *team worker*. Thus, for permissions of a *project manager* not only the axioms for *project managers* but also for *team workers* hold. For instance, referring to Example 1 and 2, a *project manager* has reading access both to his own accountings (independent of the projects) and to all accountings of his own project (independent of the user).

The second aspect is related with inheritance in the domain model referring to the objects that we want to protect with our permissions. In the same way as above, methods of subclasses inherit the permissions of their superclasses. In addition, we provide the possibility to let permissions unspecified in superclasses and deferring their specification to their subclasses.

4 Specification of Access Policies in OCL

Regarding tool support for the modelling of access rights and for implementation aspects the formal specification in the framework presented has to be transformed into a specification language, designed especially for use in the context of diagrammatic specification languages such as UML.

In the following we show the realisation of our concepts within the Object Constraint Language (OCL) [8,9] that is part of UML. We extend the specification section of a method (comprising the pre- and postcondition) by an additional *permission section*. In this section the method permission is specified by an OCL-expression using the variables of the method, the actual object and the actor which is handled as parameter of the permission section.

The representation function *rolerep* is treated as query operation of the actor hierarchy as introduced in section 3.1 and may be used in the permission section.

Example 1: A team worker can read all his own accountings, independent of the project the accounting belongs to (exemplified by the permission of *getAccountingDate*).

```
context Accounting :: getAccountingDate()
   perm (act : ACTeamWorker):
      self.user = act.rolerep()
```

Example 2: A team worker can only write own accountings of released activities.

```
context Accounting :: writeAccountingDate()
   perm (act : ACTeamWorker):
     self.user = act.rolerep() and
     self.activity.state = ActivityState::released
```

Example 3: The project manager can only create accountings for activities of own projects, i.e. activities of projects, where he is project manager of.

```
context Accounting :: create(a : Activity, u : User)
   perm (act : ACProjectManager):
     a.project.projectmanager = act.rolerep()
```

Regarding the semantics of a method permission it has to be made clear that the given actor is *not* the object directly calling the method but the role initiating the call of this method from outside the system (eventually causing a chain of method calls). In the implementation the calling actors can be handled by an additional method parameter or, like in J2EE, by some method call infrastructure.

5 Extensions

As explained in the previous section our basic view of user rights is that of an actor having permission to perform a certain method on a certain object. This fine-grained paradigm provides a maximum of flexibility for specifying any kind of user permissions in all phases of the development. However, it is clear that for practical applications we need an aggregation mechanism for supporting more coarse-grained specifications. We therefore introduce the notion of *method categories*. A method category CAT basically is a set of methods

$$CAT \subseteq METH$$

where $METH$ is the set of all methods (sorted by the classes they belong to) in the system. Categories may contain methods of a single class (we use the class name as index in this case) or of several classes. Moreover, categories may comprise other categories, i.e. categories may be structured in a hierarchical way. We define coarse-grained permissions

$$perm_{Cat_C} : (ACActor, C)\ Bool$$

for each category Cat_C containing methods of class C. A category permission induces method permissions in the obvious way.

$$\begin{gathered}\forall\, a : ACActor,\ o : C,\ a_1 : T_1, \ldots, a_n : T_n\ .\\ perm_{Cat_C}(a, o) \Rightarrow perm_{C,m}(a, o, a_1, \ldots, a_n)\end{gathered}$$

for all methods m of class C in Cat_C. Of course, such a category permission may only depend on the actor and the actual object.

If a category Cat comprises methods of different classes (and create methods) we define permissions.

$$perm_{Cat} : (ACActor)\ Bool$$

and induce the following method permissions for all methods m of class C in Cat.

$$\begin{aligned}&\forall\, a : ACActor,\; o : C,\; a_1 : T_1, \ldots, a_n : T_n \;.\\ &perm_{\,Cat}(a) \Rightarrow perm_{\,C,m}(a, o, o_1, \ldots, o_n)\end{aligned}$$

Category permissions of this kind only depend on the actor initiating the method call. Concerning the application of this concept we provide a set of predefined categories:

$READ_C$	the category of all methods reading some attribute of class C
$UPDATE_C$	the category of all methods updating some attributes of class C
$CREATE_C$	the category of all creation methods of class C

Moreover, we define the get and set methods of attributes to be predefined members of the $READ_C$ and $UPDATE_C$ category, respectively. Please notice that there may be methods both belonging to the $READ_C$ and the $UPDATE_C$ category.

As an example the informal specifications of Table 2 can be immediately expressed in a corresponding way with method categories. E.g. the clause *"The team worker can read all own accountings"* can be expressed with the following category permission:

$$\begin{aligned}&\forall\, tw : ACTeamWorker \;.\; \forall\, a : Accounting \;.\\ &a.user = tw.rolerep\,()\\ &\quad \Rightarrow perm_{\,READ_{Accounting}}(ac, a)\end{aligned}$$

The set of predefined method categories may be replaced and complemented by user-defined categories. This is advisable if a whole part of the class diagram is associated with the same kind of permissions (e.g. the permission *true*). A further typical case in which we need a more fine-grained categorisation is the following. The attributes of a class (e.g. *Person*) are divided into critical (e.g. salary of a person) and uncritical ones (e.g. name and address of a person).

6 Conclusion

In the preceding sections we introduced a formal specification framework for the modeling of user rights. Our method is novel in the respect that it is completely integrated in the concepts of use case driven object oriented modeling and provides the full expressiveness of first-order logic. We separate the aspects of authentication and data access and thus enable a concise specification of permissions connecting roles and their internal representation. We both support the specification of permissions on the most fine-grained level of methods and on a coarse-grained level based on the notion of method categories.

Currently we conduct two case studies in real contexts (health information systems, e-government) in order to validate our approach. Future work will be done in several directions. First of all we will develop tool support for our method. This comprises the possibility to develop method permissions, actor specifications and the definition of categories and category permissions within some UML tool. Moreover, our concept of specifying permissions well be applied in implementation oriented contexts. In the project SECTINO we develop a framework for specifying access policies for inter-organisational workflows based on Web Services. Besides this, we work on a testing environment testing and analysing rights in collaborative systems.

References

1. Ferraiolo, D.F., Chandramouli, R., Kuhn, D.R.: Role-Based Access Control. first edn. Artech House Publishers (2003)
2. Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST Standard for Role-Based Access Control. In: ACM Transactions on Information and System Security. Number 3. ACM (2001) 224–274 http://csrc.nist.gov/rbac/rbacSTD-ACM.pdf.
3. Sandhu, R.S.: Role Hierarchies and Constraints for Lattice-Based Access Controls. In: Proceedings of the European Symposium on Research in Security and Privacy. (1996)
4. Miller, J., Fan, M., Sheth, A.P., Kochut, K.: Security in Web-Based Workflow Management Systems. In: Proceedings of the International Workshop on Research Directions in Process Technology, Nancy, France (1997)
5. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison Wesley Longman, Inc. (1999)
6. http://www.v-modell.iabg.de.
7. Breu, R., Burger, K., Hafner, M., Jürjens, J., Popp, G., Wimmel, G., Lotz, V.: Key Issues of a Formally Based Process Model for Security Engineering. In: Proceedings of the 16th International Conference on Software & Systems Engineering and their Applications (ICSSEA03), Paris, December 2 - 4, 2003. (2003)
8. Warmer, J., Kleppe, A.G.: The Object Constraint Language – Precise Modeling with UML. first edn. Addison Wesley Longman, Inc. (1999)
9. OMG: Unified Modeling Language Specification – Version 1.5 (2003)
10. Breu, R.: An Integrated Approach to Use Case Based Development (2004) To appear.
11. Breu, R.: Objektorientierter Softwareentwurf – Integration mit UML. Springer-Verlag (2001) in German.
12. Lodderstedt, T., Basin, D., Doser, J.: SecureUML: A UML-based Modeling Language for Model-Driven Security. In: Proceedings LNCS 2460, Springer (2002) 426–441
13. Fernandez, E., Hawkins, J.: Determining role rights from use cases. In: Workshop on Role-Based Access Control, ACM (1997) 121–125
14. Höhn, S., Jürjens, J.: Automated Checking of SAP Security Permissions. In: Proceedings of the 6th IFIP WG 11.5 Working Conference on Integrity and Internal Control in Information Systems (IICIS), Nov. 13-15, 2003, Lausanne, Switzerland, Kluwer (2003)
15. Services, I.B.C.: SAP Berechtigungswesen, Design und Realisierung von Berechtigungskonzepten f r SAP R/3 und SAP Enterprise Portal. SAP Press (2003) in German.

Modeling Role-Based Access Control Using Parameterized UML Models

Dae-Kyoo Kim, Indrakshi Ray, Robert France, and Na Li

Computer Science Department
Colorado State University
Fort Collins, CO 80523, USA
{dkkim,iray,france,na}@cs.colostate.edu
Fax: +1 970 491 2466

Abstract. Organizations use Role-Based Access Control (RBAC) to protect computer-based resources from unauthorized access. There has been considerable work on formally specifying RBAC policies but there is still a need for RBAC policy specification techniques that can be integrated into software design methods. This paper describes a method for incorporating specifications of RBAC policies into UML design models. Reusable RBAC policies are specified as patterns and are expressed using UML template diagrams. Incorporating RBAC policies into an application specific model involves instantiating the patterns and composing the instantiations with the model. The method also includes a technique for specifying patterns of RBAC violations. Developers can use the patterns to identify policy violations in their models. The method is illustrated using a small banking application.

1 Introduction

Access control policies are constraints that protect computer-based information resources from unauthorized access. Role-Based Access Control (RBAC) [8] is used by many organizations to protect their information resources from unauthorized access. RBAC policies are defined in terms of permissions that are associated with roles assigned to users. A permission determines what operations a user assigned to a role can perform on information resources.

Work on formalization of RBAC policies has resulted in the development of new specification notations (e.g., see [1]), but there is still a need for policy specification approaches that can be integrated with design techniques used in industry. The Unified Modeling Language (UML) is considered to be the industry de-facto standard for modeling software-based systems. Use of the UML to specify RBAC policies eases the task of incorporating the policies into UML application models.

This paper describes a method that integrates RBAC policy specification and UML design modeling. The method includes (1) a technique for specifying generic RBAC policies as patterns that can be instantiated to produce application-specific design structures that specify the RBAC constraints, (2) techniques for systematically incorporating design structures produced by RBAC policy patterns into UML design models, and (3) a technique for specifying design structures that violate RBAC constraints as patterns.

Generic RBAC policies are specified by patterns expressed as UML diagram templates. Instantiating a RBAC pattern to produce an application-specific RBAC design structure involves binding the template parameters to application-specific design elements.

The RBAC patterns can be used to support at least two approaches to incorporating RBAC constraints into a UML design model: (1) The templates can be used to produce an initial design structure that is then extended to address other design concerns; and (2) the design structures produced by the templates can be merged with a previously developed application design model, referred to as the *primary model*, to obtain a design model that specifies RBAC constraints.

The method also provides a technique for specifying RBAC constraint violations as patterns. The violation patterns are expressed as template object diagrams, and can be used to check for the presence of violations in designs. To ease the task of checking for violations using the patterns we have developed an approach to visualizing application-specific RBAC constraints as object diagrams.

An overview of RBAC is given in Section 2. In Section 3 we present a generic RBAC model expressed as a class diagram template, and give examples of object diagram templates that describe patterns of policy violations. Section 4 describes how the RBAC pattern can be incorporated into a primary model. Section 5 gives examples of application-specific RBAC policies expressed as object diagrams. Section 6 illustrates how the violation patterns described by object diagram templates can be used to detect violations in application-specific RBAC policies. An overview of related work is provided in Section 7. The paper concludes with a discussion of current limitations of the approach and our plans to address the limitations.

2 Overview of RBAC

RBAC constraints can be organized as follows: *Core RBAC*, *Hierarchical RBAC*, *Static Separation of Duty Relations*, and *Dynamic Separation of Duty Relations*.

Core RBAC embodies the essential aspects of RBAC. The constraints specified by Core RBAC are present in any RBAC model. The Core RBAC requires that users be assigned to roles (job function), roles be associated with permissions (approval to perform an operation on an object), and users acquire permissions by being assigned to roles. The Core RBAC does not place any constraint on the cardinalities of the user-role assignment relation or the permission-role association. Core RBAC also includes the notion of user sessions. A user establishes a session during which he activates a subset of the roles assigned to him. Each user can activate multiple sessions; however, each session is associated with only one user. The operations that a user can perform in a session depend on the roles activated in that session and the permissions associated with those roles.

Hierarchical RBAC adds features supporting role hierarchies. Hierarchies are used to describe a structure of roles in an organization. Role hierarchies define an inheritance relation among the roles. Role $r1$ inherits from role $r2$ only if all permissions of $r2$ are also permissions of $r1$ and all users of $r1$ are also users of $r2$. The inheritance relationship is reflexive, transitive and anti-symmetric.

Static Separation of Duty (SSD) relations are necessary to prevent conflict of interests that arise when a user gains permissions associated with conflicting roles (roles that cannot be assigned to the same user). SSD relations are specified for any pair of roles that conflict. The SSD relation places a constraint on the assignment of users to roles, that is, assignment to a role that takes part in an SSD relation prevents the user from being assigned to the related conflicting role. The SSD relationship is symmetric, but it is neither reflexive nor transitive. SSD may exist in the absence of role hierarchies (referred to as SSD RBAC), or in the presence of role hierarchies (referred to as hierarchical SSD RBAC). The presence of role hierarchies complicates the enforcement of the SSD relations: before assigning users to roles not only should one check the direct user assignments but also the indirect user assignments that occur due to the presence of the role hierarchies.

Dynamic Separation of Duty (DSD) relations aim to prevent conflict of interests as well. The DSD relations place constraints on the roles that can be activated in a user's session. If one role that takes part in a DSD relation is activated, the user cannot activate the related (conflicting) role in the same session. A model of RBAC is shown in Fig. 1.

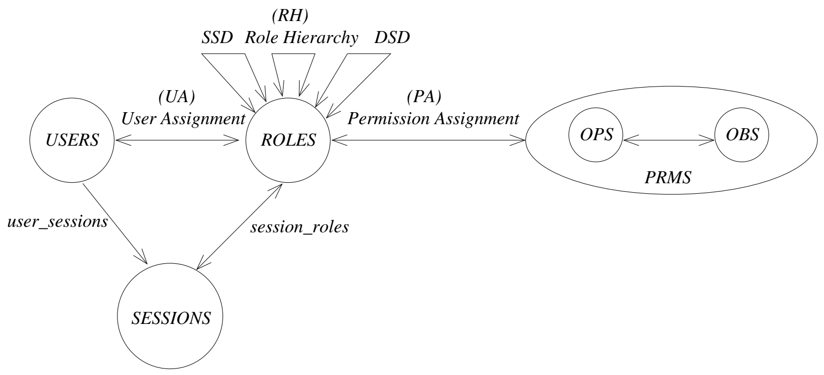


Fig. 1. RBAC.

The RBAC in Fig. 1 consists of: 1) a set of users ($USERS$) where a user is an intelligent autonomous agent, 2) a set of roles ($ROLES$) where a role is a job function, 3) a set of objects (*OBS*) where an object is an entity that contains or receives information, 4) a set of operations (*OPS*) where an operation is an executable image of a program, and 5) a set of permissions (*PRMS*) where a permission is an approval to perform an operation on objects. The cardinalities of the relationships are indicated by the absence (denoting one) or presence of arrows (denoting many) on the corresponding associations. For example, the association of user to session is one-to-many. All other associations shown in the figure are many-to-many. The association labeled *Role Hierarchy* defines the inheritance relationship among roles. The association labeled *SSD* specifies the roles that conflict with each other. The association labeled *DSD* specifies the roles that cannot be activated within a session by the same user.

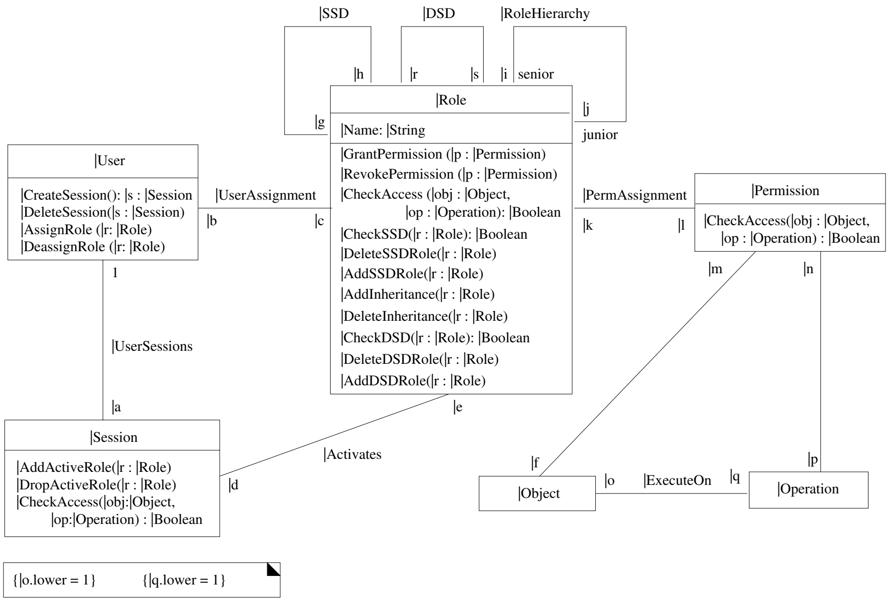

|SSD
|DSD
|RoleHierarchy
|h
|r
|s
|i
senior
|g
|j
junior
|Role
|Name: |String
|GrantPermission (|p : |Permission)
|RevokePermission (|p : |Permission)
|CheckAccess (|obj : |Object, |op : |Operation): |Boolean
|CheckSSD(|r : |Role): |Boolean
|DeleteSSDRole(|r : |Role)
|AddSSDRole(|r : |Role)
|AddInheritance(|r : |Role)
|DeleteInheritance(|r : |Role)
|CheckDSD(|r : |Role): |Boolean
|DeleteDSDRole(|r : |Role)
|AddDSDRole(|r : |Role)
|User
|CreateSession(): |s : |Session
|DeleteSession(|s : |Session)
|AssignRole (|r: |Role)
|DeassignRole (|r: |Role)
|UserAssignment
|b
|c
|PermAssignment
|k
|l
|Permission
|CheckAccess(|obj : |Object, |op : |Operation) : |Boolean
1
|UserSessions
|a
|m
|n
|e
|Session
|AddActiveRole(|r : |Role)
|DropActiveRole(|r : |Role)
|CheckAccess(|obj:|Object, |op:|Operation) : |Boolean
|Activates
|d
|f
|Object
|o
|ExecuteOn
|q
|p
|Operation
{|o.lower = 1}
{|q.lower = 1}

Fig. 2. A RBAC Class Diagram Template.

3 A Reusable RBAC Model

In this section a RBAC pattern is described as a UML template class diagram. A class diagram is obtained from a template diagram by binding the parameters to values. Fig. 2 shows a class diagram template describing hierarchical RBAC with SSD and DSD. The symbol "|" is used to indicate parameters. We use this notation when there is a large number of parameters because the standard UML parameter notation is cumbersome.

The class diagram template shown in Fig. 2 consists of class and association templates. A class template is a class descriptor with parameters. Class templates are associated with attribute templates (e.g., $|Name : String$ in *Role*) and operation templates (e.g., $|GrantPermission$ in *Role*). Association templates (e.g., $|UserAssignment$) consist of parameters for association names and association-end multiplicities. The OCL constraints in Fig. 2 restrict the values that can be bound to association-end multiplicity parameters. For example, $\{|o.lower = 1\}$ restricts the multiplicities that can be bound to the parameter *o* to ranges that have a lower bound of 1. The multiplicity "1" on the $UserSessions$ association-end attached to $User$ is strict: a session can only be associated with one user.

The *User* class template defines classes that describe users. A user can create a new session (*CreateSession*), delete a session (*DeleteSession*), associate self with a new role (*AssignRole*) and remove an associated role (*DeassignRole*). A *UserSessions* link (i.e., an instance of an association obtained by binding the parameters of $UserSessions$ to values) is created by a *CreateSession* operation (i.e., an operation obtained by binding the operation template parameters to values) and deleted by a *DeleteSession* operation. The operation *AssignRole* creates a *UserAssignment* link; the *DeassignRole* removes a *UserAssignment* link.

The class template *Role* is used to produce classes representing roles with behavior that (1) associates a new permission with the role (*GrantPermission*), (2) deletes an existing permission associated with the role (*RevokePermission*), (3) adds an immediate inheriting role (*AddInheritance*), (4) deletes an immediate inheriting role (*DeleteInheritance*), (5) adds a role to the set of conflicting roles (*AddSSDRole*), (6) deletes a role from the existing set of conflicting roles (*DeleteSSDRole*), (7) checks whether the role is in an SSD relationship with a given role in the presence of hierarchies (*CheckSSD*), (8) checks whether the role has a given permission (*CheckAccess*), (9) checks whether the role is in a DSD relation with a given role (*CheckDSD*), (10) deletes a DSD relation between the role and a given role (*DeleteDSDRole*), and (11) adds a DSD relation with a given role (*AddDSDRole*). The class template *Session* is associated with the template operations: *AddActiveRole* (activates a role in a session), *DropActiveRole* (deactivates a role in a session), and *CheckAccess* (checks whether the role has the permission to perform an operation on an object).

The class template *Permission* is associated with an operation template, *CheckAccess*, that checks whether the role has the permission to perform the operation on the object.

Each operation template is associated with an OCL template expression that produces OCL pre- and post-conditions when the template parameters are bound to values. Pre- and post-condition templates associated with the *CreateSession* and *GrantPermission* operation templates are given below:

```
context |User::|CreateSession():(|s:|Session)
   post: result = |s and
    |s.oclIsNew() = true and self.|Session → includes(|s)

context |Role::|GrantPermission (|p:|Permission)
   post: self.|Permission → includes(|p)
```

We express RBAC constraints that restrict SSD and DSD relationships as OCL template expressions. Examples of these constraints are given below:

– SSD constraint. A user cannot be assigned to two roles that are involved in an SSD relation.

```
context |User inv:
    self.|Role → forAll(r1, r2 | r1.|SSD → excludes(r2))
```

– Hierarchical SSD constraint. There cannot be roles in an SSD relation which have the same senior role.

```
context |Role inv:
let allSenior(r1) = r1.senior → union(r1.senior → collect(r2 | allSenior(r2)))
in
    self.|SSD → forAll(r1 | allSenior(r1) → excludesAll(allSenior(self))
```

– DSD constraint. A user cannot activate two roles in DSD relation within a session.

```
context |User inv:
    |self.|Session.|Activates → forAll(r1, r2 | r1.|DSD → excludes(r2))
```

4 Applying the RBAC Model

To illustrate our approach we use a simple banking application taken from [5]. The application is used by various bank officers to perform transactions on customer deposit accounts, customer loan accounts, ledger posting rules, and general ledger reports. The transactions include 1) create, delete, or modify customer deposit accounts, 2) create, delete, or modify customer loan accounts, 3) modify the ledger posting rules, and 4) create general ledger report. A class diagram (the primary model) for the application is shown in Fig. 3. Class attributes and operations are not shown in the diagram.

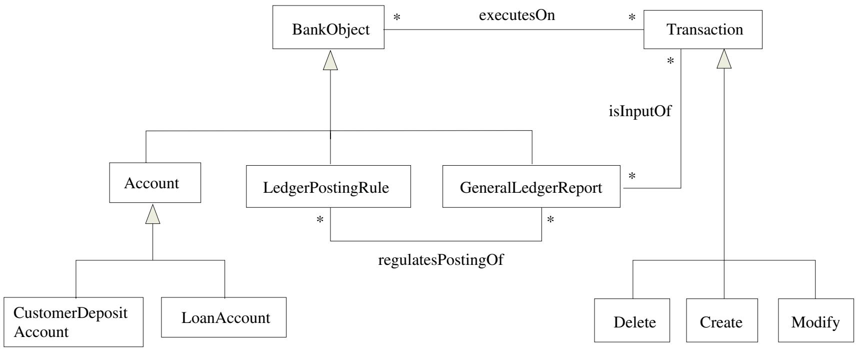


Fig. 3. A Partial View of a Banking System Primary Model.

Access control policies are not specified in the primary model. RBAC features can be incorporated into the primary model by composing an instantiation of the RBAC template in Fig. 2 with the primary model. The composition is carried out as follows:

1. Instantiating the RBAC template: To incorporate RBAC features into a primary model, the modeler must first instantiate the RBAC template model by binding parameters to elements representing concepts in the domain of the primary model. Some of these model elements may be elements in the primary model. Class diagrams obtained from the RBAC class diagram template are referred to as *context-specific RBAC diagrams*. Fig. 4 shows a context-specific RBAC class diagram for the banking application. In the diagram, *BankRole*, *BankObject*, and *Transaction* are respectively bound to *Role*, *Object*, and *Operation* parameters in the RBAC template diagram.

2. Merging the context-specific diagram with the primary model: The view defined by the context-specific RBAC diagram is merged with the view defined in the primary model to obtain a composed model. Elements in the instance and the primary model are merged if and only if they have the same syntactic type (i.e., UML metamodel class) and name. Model elements in the context-specific RBAC diagram that do not exist in the primary model are added to the primary model. For example, if the RBAC diagram has a class with an operation that has the same name, but different signature as a primary

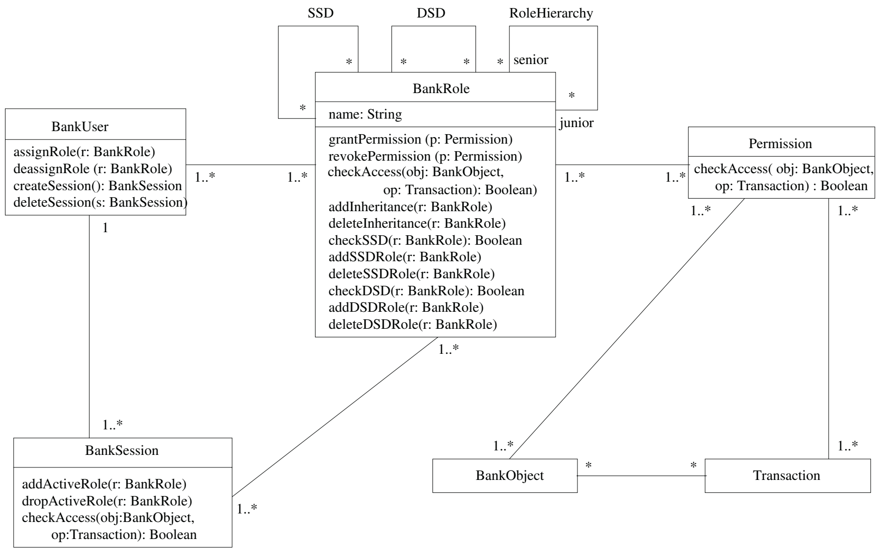

SSD
DSD
RoleHierarchy
senior
junior
BankRole
name: String
grantPermission (p: Permission)
revokePermission (p: Permission)
checkAccess(obj: BankObject, op: Transaction): Boolean)
addInheritance(r: BankRole)
deleteInheritance(r: BankRole)
checkSSD(r: BankRole): Boolean
addSSDRole(r: BankRole)
deleteSSDRole(r: BankRole)
checkDSD(r: BankRole): Boolean
addDSDRole(r: BankRole)
deleteDSDRole(r: BankRole)
BankUser
assignRole(r: BankRole)
deassignRole (r: BankRole)
createSession(): BankSession
deleteSession(s: BankSession)
Permission
checkAccess(obj: BankObject, op: Transaction) : Boolean
BankSession
addActiveRole(r: BankRole)
dropActiveRole(r: BankRole)
checkAccess(obj:BankObject, op:Transaction): Boolean
BankObject
Transaction
1
1..*
*

Fig. 4. A Context-Specific RBAC Class Diagram.

model operation in a matching class, the operation is included in the composed model, resulting in an overloaded operation. However, if the RBAC diagram has a class with an operation with the same name and signature as a primary model operation in a matching class, either one of the operation names must be changed or the operation specifications must be logically composed to produce a consistent operation specification. In the first case, the developer must specify the new name of the operation and in the second case the developer must specify the logical operator to be used in the composition. These developer inputs are expressed as *composition directives*: A directive allows a developer to vary how RBAC and primary model elements are merged. For more on merging rules and composition directives see [9, 10].

The result of the composition is a composed model in which access control features specified by the context-specific RBAC model are incorporated into the primary model. The composed model for the banking system is shown in Fig. 5. The *BankObject* and *Transaction* classes in the context-specific RBAC diagram are merged with *BankObject* and *Transaction* classes in the primary model, and *BankUser*, *BankRole*, *BankSession*, and *Permission* are the RBAC classes that are included in the composed model.

5 Describing Application-Specific RBAC Policies Using Object Diagrams

Application-specific RBAC policies constrain how system users access system resources. They determine 1) the assignment of roles to system users, 2) the permissions associated with roles in the systems, 3) the inheritance relationships between roles, and 4) the SSD and DSD relationships between roles. In this section we illustrate how application-specific RBAC policies can be described by object diagrams.

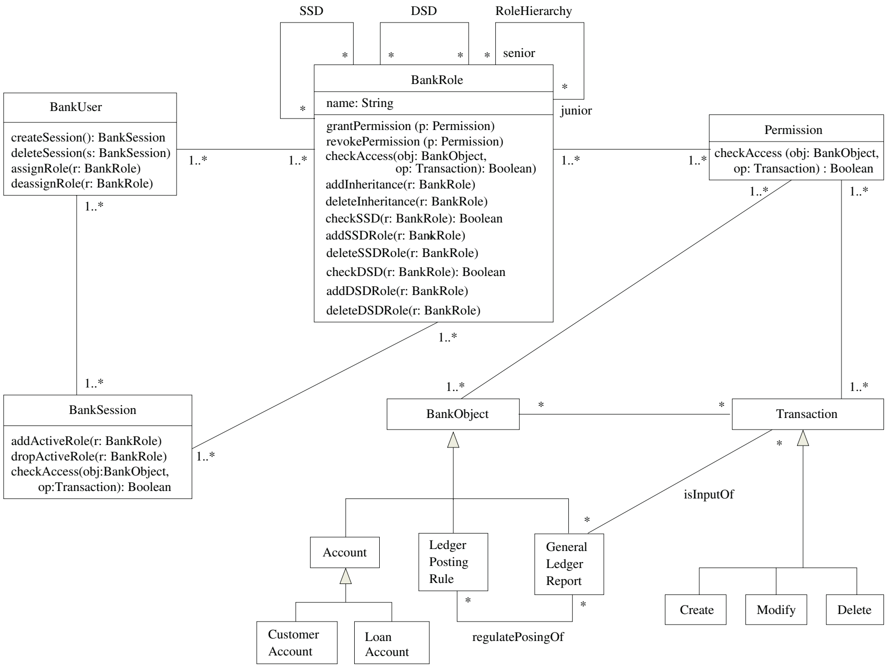

SSD
DSD
RoleHierarchy
senior
junior
BankRole
name: String
grantPermission (p: Permission)
revokePermission (p: Permission)
checkAccess(obj: BankObject, op: Transaction): Boolean)
addInheritance(r: BankRole)
deleteInheritance(r: BankRole)
checkSSD(r: BankRole): Boolean
addSSDRole(r: BankRole)
deleteSSDRole(r: BankRole)
checkDSD(r: BankRole): Boolean
addDSDRole(r: BankRole)
deleteDSDRole(r: BankRole)
BankUser
createSession(): BankSession
deleteSession(s: BankSession)
assignRole(r: BankRole)
deassignRole(r: BankRole)
Permission
checkAccess (obj: BankObject, op: Transaction) : Boolean
BankSession
addActiveRole(r: BankRole)
dropActiveRole(r: BankRole)
checkAccess(obj:BankObject, op:Transaction): Boolean
BankObject
Transaction
isInputOf
Account
Ledger Posting Rule
General Ledger Report
Customer Account
Loan Account
regulatePosingOf
Create
Modify
Delete
1..*
*

Fig. 5. The Composed Model.

The RBAC model supports the specification of four types of policies: 1) *core policies* that conform to core RBAC, that is, policies that determine user-role and role-permission assignments, 2) *hierarchical policies* that conform to hierarchical RBAC, that is, policies that determine inheritance relationships between roles, 3) *SSD policies* that conform to SSD RBAC, that is, policies that determine what roles are conflicting, and 4) *DSD policies* that conform to DSD RBAC, that is, policies that determine what roles to be activated in a session. A set of application-specific RBAC policies for the banking system is given below:

Core Policies: The roles of the banking system (instances of *BankRole*) are *teller*, *customerRerviceRep*, *accountant*, *accountingManager* and *loanOfficer*. The permissions assigned to these roles are given below:

P1 A teller can modify customer deposit accounts.
P2 A customer service representative can create or delete customer deposit accounts.
P3 An accountant can create general ledger reports.
P4 An accounting manager can modify ledger-posting rules.
P5 A loan officer can create and modify loan accounts.

Fig. 6 shows the object diagrams describing policies P1 to P5 respectively.

Hierarchical Policies: A role hierarchy defines inheritance relationships between roles. Through the inheritance relationship, a senior role inherits the permissions of its junior roles and any user assigned to the senior role is also assigned to the junior roles. The hierarchical policies in the banking application are stated below:

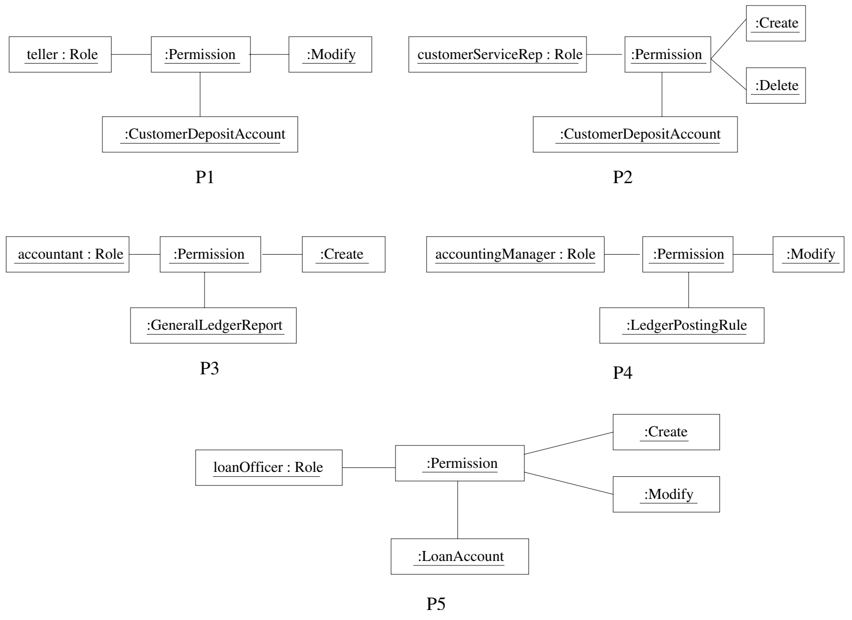


Fig. 6. Object Diagrams describing Core RBAC Policies.

H1 Customer service representative role is senior to the teller role.
H2 Accounting manager role is senior to the accountant role.

Fig. 7(a),(b) describe policies H1 and H2 respectively.
SSD Policies: SSD policies prevent a user from being assigned to two conflicting roles. For the banking system the following pairs of roles are conflicting:
$\{$*(teller, accountant), (teller, loanOfficer), (loanOfficer, accountant), (loanOfficer, accountingManager), (customerServiceRep, accountingManager)*$\}$ The object diagram in Fig. 8 describes the SSD RBAC policies.

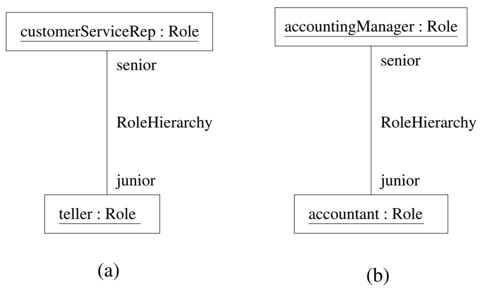


Fig. 7. Object Diagrams for Hierarchical Policies.

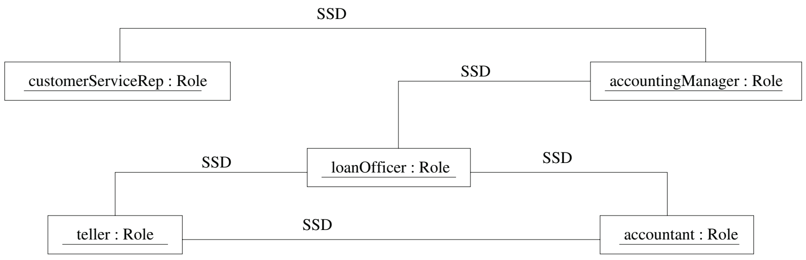


Fig. 8. Object Diagram for SSD Policies.

DSD Policies: DSD policies prevent a user from playing a role in a session, if another role in a DSD relation has been activated. For the banking system the following pair of roles are in DSD relation:
{*(customerServiceRep, loanOfficer)*} The object diagram in Fig. 9 describes the DSD RBAC policy.

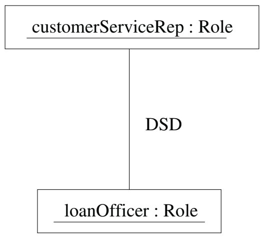
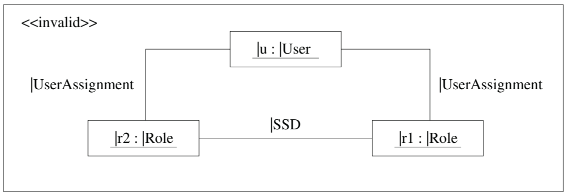


Fig. 9. Object Diagram for DSD Policy.

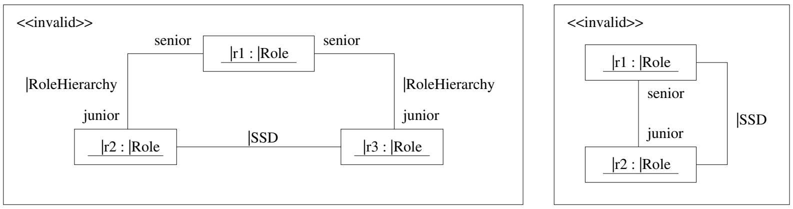
6 Identifying Conflicts in Application-Specific RBAC Policies

In this section we show how RBAC violation patterns expressed as object diagram templates can be used to identify conflicts in application-specific policies. If a violation pattern exists in an object diagram describing an application-specific policy, then a conflict exists.

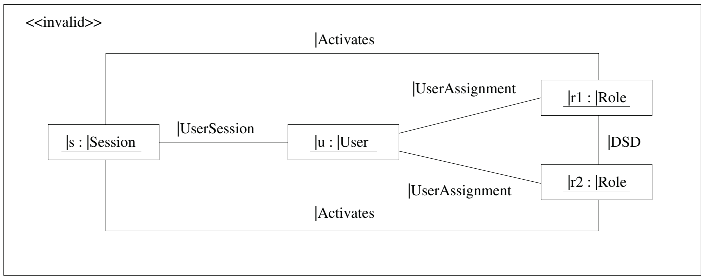
Fig. 10 shows object diagram templates that when instantiated produce object structures that violate RBAC constraints. Fig. 10(a) describes structures in which a user is assigned to roles in an SSD relationship (violation of the SSD constraint). Fig. 10(b) describes structures in which two roles in an SSD relationship have a common senior role and structures in which a senior role is in an SSD relationship with a junior role (both are violations of the hierarchical SSD constraint). Fig. 10(c) describes structures in which a user in a session activates two roles that are in a DSD relationship (a violation of the DSD constraint). We illustrate how these object diagram templates can be used to identify conflicts in application models later in this paper.



(a) Violation of SSD Constraint



(b) Violations of Hierarchial SSD Constraint



(c) Violation of DSD Constraint

Fig. 10. RBAC Constraints.

Fig. 11 shows the object diagram that integrates the policies shown in Fig. 7, Fig. 8, and Fig. 9. The reader can visually check that the pattern described by object diagram template in Fig. 10(b) does not occur in Fig. 11.

Formally, an object diagram has the violation described by a violation pattern if there exists a binding that produces an object structure contained in the object diagram. To illustrate how conflicts can be identified, consider the case in which the following policy is added to the set of policies described in the previous section: “The branch manager role is senior to all the other roles in the bank.” Fig. 12 shows the result of including this policy in the banking application’s policy set. A number of occurrences of the pattern described in Fig. 10(b) can be found in Fig. 12. For example, if we assign a user to the branch manager role, the user is also assigned to the roles *customerServiceRep* and *accountingManager* through inheritance. However, the roles *customerServiceRep* and *accountingManager* are in an SSD relation.

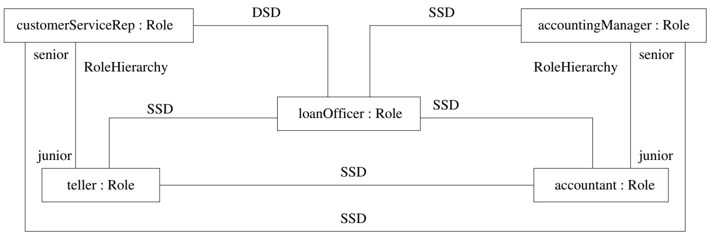


Fig. 11. Combined Object Diagram.

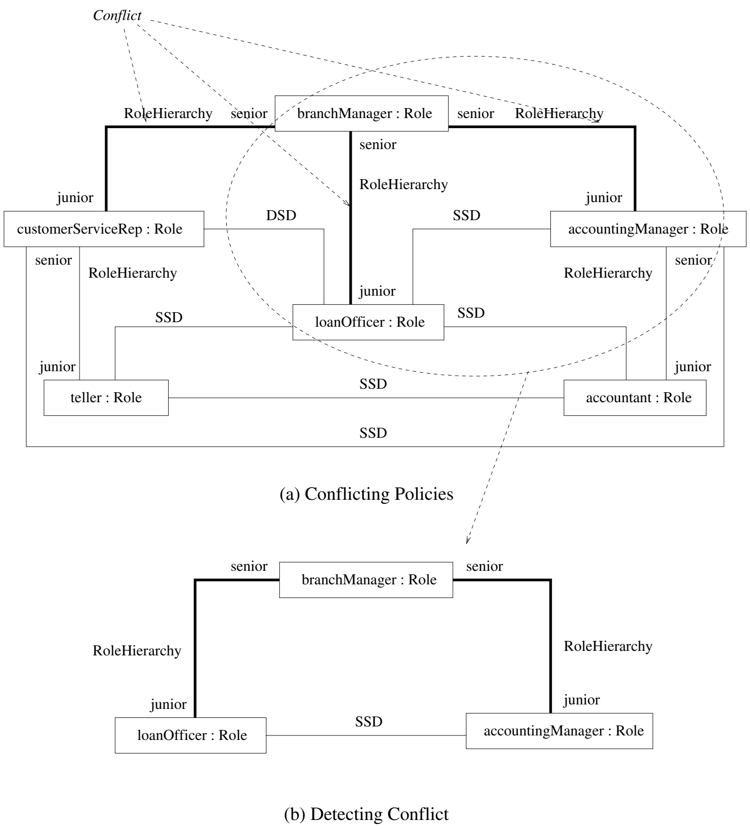


Fig. 12. Violation Pattern Occurrence: Hierarchical SSD.

7 Related Work

Tidswell and Jaeger [21] propose an approach to visualizing access control constraints. They point out the need for visualizing constraints and the limitations of previous work on expressing constraints. A drawback of their work is that they created a new notation for specifying constraints and it is not clear how the new notation can be integrated with other widely-used design notations. The approach described in this paper utilizes a popular standardized modeling language (the UML) and also integrates the policy specification activity with UML design modeling activities.

A large volume of research (e.g., see [2–4, 6, 7, 11, 12, 14] exists in the area of access control policy specification. Formal logic-based techniques (e.g., see [2–4, 6, 11, 14]) are often used to specify security policies. The use of mathematical concepts and notation that are not familiar to software developers makes them difficult to use and understand. Other researchers have used high-level languages to specify policies [12, 13, 19, 20]. Although high-level languages are easier to understand than formal logic-based approaches, they are not analyzable.

Some work has been done on modeling system security using UML. Jurjens [15] proposes UMLsec, a UML profile for modeling and evaluating security aspects based on the multi-level security model. Lodderstedt *et al.* propose SecureUML [17], an extension of the UML that defines security concepts based on RBAC. These approaches mainly focus on extending the UML notation to better reflect security concerns. The approach described in this paper tackles the complementary task of capturing RBAC policies in patterns that can be reused by developers of secure systems.

8 Conclusion

The work described in this paper focuses on specifying only the static structure of RBAC. A complete RBAC model should also include descriptions of the patterns of behavior supported by RBAC. In previous work (e.g., see [9, 16]) we developed template forms of interaction diagrams that can be used to specify interaction patterns. The interaction patterns can be used to characterize families of allowed and prohibited behaviors. We are also developing template forms of other UML behavioral models.

The use of violation patterns to identify policy conflicts, while useful, has its limitations. Checking for the presence of a pattern in an object diagram specifying a set of policies is essentially a search for a subgraph in an object diagram, which is known as *subgraph isomorphism problem*. Detecting subgraph isomorphism can be a difficult task [18]. Our work in this area focuses on identifying alogorithms to support practical application of this technique.

Validation of the method is needed. To support planned validation activities we are developing a tool set that allows developers to create and instantiate UML diagram templates, and to compose template instantiations with UML design models.

References

1. G.J. Ahn and R. Sandhu. Role-based Authorization Constraints Specification. *ACM Transactions on Information and Systems Security*, 3(4):207–226, November 2000.
2. S. Barker. Security Policy Specification in Logic. In *Proceedings of the International Conference on Artificial Intelligence*, pages 143–148, Las Vegas, NV, 2000.
3. S. Barker and A. Rosenthal. Flexible Security Policies in SQL. In *Proceedings of the 15th Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, Niagara-on-the-Lake, Canada, 2001.

4. E. Bertino, P. Bonatti, and E. Ferrari. TRBAC: A Temporal Role-Based Access Control Model. In *Proceedings of the 5th ACM Workshop on Role-Based Access Control*, pages 21–30, Berlin, Germany, 2000.
5. R. Chandramouli. Application of XML Tools for Enterprise-Wide RBAC Implementation Tasks. In *Proceedings of 5th ACM workshop on Role-based Access Control*, Berlin, Germany, July 2000.
6. F. Chen and R. Sandhu. Constraints for Role-Based Access Control. In *Proceedings of the 1st ACM Workshop on Role-Based Access Control*, Gaithersburg, MD, 1995.
7. N. Damianou and N. Dulay. The Ponder Policy Specification Language. In *Proceedings of the Policy Workshop*, Bristol, U.K., 2001.
8. D.F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and Systems Security*, 4(3), August 2001.
9. Geri Georg, Robert France, and Indrakshi Ray. An Aspect-Based Approach to Modeling Security Concerns. In *Proceedings of the Workshop on Critical Systems Development with UML*, Dresden, Germany, 2002.
10. Geri Georg, Indrakshi Ray, and Robert France. Using Aspects to Design a Secure System. In *Proceedings of the Interational Conference on Engineering Complex Computing Systems (ICECCS 2002)*, Greenbelt, MD, December 2002. ACM Press.
11. R. J. Hayton, J. M. Bacon, and K. Moody. Access Control in Open Distributed Environment. In *IEEE Symposium on Security and Privacy*, pages 3–14, Oakland, CA, May 1998.
12. M. Hitchens and V. Varadarajan. Tower: A Language for Role-Based Access Control. In *Proceedings of the Policy Workshop*, Bristol, U.K., 2001.
13. J. A. Hoagland, R. Pandey, and K. N. Levitt. Security Policy Specification Using a Graphical Approach. Technical Report CSE-98-3, Computer Science Department, University of California Davis, July 1998.
14. S. Jajodia, P. Samarati, and V. S. Subrahmanian. A Logical Language for Expressing Authorizations. In *IEEE Symposium on Security and Privacy*, pages 31–42, Oakland, CA, May 1997.
15. J. Jurjens. UMLsec: Extending UML for Secure Systems Development. In *Proceedings of Fifth International Conference on the Unified Modeling Language, pp. 412-425*, pages 412–425, Dresden, Germany, October 2002.
16. Dae-Kyoo Kim, Robert France, Sudipto Ghosh, and Eunjee Song. Using Role-Based Modeling Language (RBML) as Precise Characterizations of Model Families. In *Proceedings of the Interational Conference on Engineering Complex Computing Systems (ICECCS 2002)*, Greenbelt, MD, December 2002. ACM Press.
17. T. Lodderstedt, D. A. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *Proceedings of Fifth International Conference on the Unified Modeling Language*, pages 426–441, Dresden, Germany, October 2002.
18. B.T. Messmer and H. Bunke. Subgraph Isomorphism in Polynomial Time. In *Lecture Notes in Computer Science Graph Theory - ECCV'98, Springer-Verlag*, Berlin, 1998.
19. OASIS. XACML Language Proposal, Version 0.8. Technical report, Organization for the Advancement of Structured Information Standards, January 2002. Available electronically from http://www.oasis-open.org/committees/xacml.
20. C. Ribeiro, A. Zuquete, and P. Ferreira. SPL: An Access Control Language for Security Policies with Complex Constraints. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2001.
21. J. E. Tidswell and T. Jaeger. An Access Control Model for Simplifying Constraint Expression. In *Proceedings of 7th ACM conference on Computer and communications security*, pages 154–163, Athens, Greese, November 2000.

Compositional Nested Long Running Transactions

Laura Bocchi

Dipartimento di Scienze dell'Informazione, University of Bologna
Mura Anteo Zamboni 7, 40127 Bologna, Italy
`bocchi@cs.unibo.it`
Fax:+39 051 2094510

Abstract. Web Services offer a widespread standard for making services available on the Internet. Of particular interest is the possibility of composing existing distributed services to create new complex ones. Existing research has already studied long running transactions within a formal context. In this other research, compensations are just partly compositional: a transaction's failure triggers the compensation of immediately enclosed transactions, but not those of nested transactions. In this paper we formally model a more compositional protocol with the asynchronous pi calculus. The resulting behavior is similar to that of the Business Transaction Protocol of OASIS [1], which also has arbitrary nesting.

1 Introduction

Web Services offer a widespread standard for making services available on the Internet, not just to humans but also to other services. Of particular interest is the possibility of composing existing distributed services, perhaps from different companies, to create a new complex service. In this sense each service is a peer that can behave both as a client and as a service provider. Current work, under the general heading 'Choreography', attempts to standardise this possibility of nested composition. Some examples of proposed standards are BPML by bpmi.org [2], XLANG by Microsoft [3], WSFL by IBM [4] and BPEL4WS by a consortium [5]. The W3C Choreography Group is currently working on the Recommendation for Web Services Choreography (a draft has been public since August 2003 [6]).

Within this context, where the parts are loosely coupled and not always trusted, standard ACID transactions (with properties of Atomicity, Consistency, Isolation, Durability) are too strict. This is especially a problem in business transactions: for instance, some 'pay-employee' transaction must be executed promptly at the start of the month, even if other necessary sub-transactions have not yet finished. The payment might subsequently be undone, 'compensated', if it turned out premature. The cycle of perform-then-maybe-compensate is one characteristic of loosely-coupled long-running transactions. Long running transactions have been introduced in 'data processing applications' [7,8], where they were called *Sagas*. Then Web Services led to a renewed interest in long running transactions that are supported, in a mainly local perspective, by already mentioned languages (WSFL, XLANG and BPEL). Other contributions arise in the context of Web transaction protocols, where loosely coupled Web services are coordinated as autonomous entities by means of a defined set of transaction messages. We mention the W3C Tentative Hold Protocol (THP) [9], OASIS BTP [1] and WS-Transactions [10] by BEA, IBM and Microsoft.

There is general agreement on the importance of such weaker transactions, but not yet an agreement on their exact meaning. In this paper we choose one particular form of weak transactions, express it in the pi calculus, and prove formally its correctness. In particular, correctness refers to deadlock absence (*Eventuality*), *Durability* and partial *Atomicity*. There is no global Atomicity in the whole set of transactions, but if a transaction fails then all its sub-transaction fail. We will discuss more later.

1.1 Related Works

Long running transactions have been described within several formal contexts. As regards XLANG (used in the product Microsoft BizTalk), its transactional behavior is informally described in [11], and then implemented in the Join calculus. In [12] the transactional behavior is formally specified at high level, and then implemented in the asynchronous pi calculus. In these other works, compensations are just partly compositional: a transaction's failure triggers the compensation of the enclosed transactions, but not those of nested transactions. It is possible to encode the effect of nested triggers, basically though copying a child's compensation code into the parent. But this is no longer compositional, and is clearly inappropriate in a Web Service context where nested transactions might belong to different (untrusted) companies. The need for compositional nested transactions is stressed in the current W3C draft of standards for Choreography and Coordination [6,10]. The transactions that we encode are compositional: a failure is able to trigger all the compensations of all its nested transactions.

The issue of the paper is describing Web transaction protocols, which are based upon message exchange in a distributed setting. We do this with the pi calculus – it is a message-based formalism, and seems natural for representing distributed protocols in the sense that it is easy to obtain a straightforward implementation. An alternative would be to use formalisms that express properties as predicates between states, such as TLA [14] or ACTA [13] (a first-order logic-based formalism for describing transactional models). These may possibly lead to a more elegant representation than using the pi calculus, but would probably no longer be as close to an implementation.

1.2 Structure of the Paper

Section 2 provides a minimal background on the pi calculus and introduces basics on the described transaction behavior. Section 3 presents an implementation of transaction managers with the asynchronous pi calculus. Section 4 formally defines some properties, i.e. Durability, Eventuality and Local Atomicity that are then proved for the given implementation. Section 5 contains some conclusive remarks.

2 Preliminaries

We will specify the BTP in the pi calculus. What follows is a brief introduction on the pi calculus, for a full reference see [15].

The asynchronous pi calculus assumes distributed entities called *processes* which exchange messages over channels, named $u, v, \ldots, z$. The content of a message is also a channel name. A process can send a message z along a channel u with the non-blocking output action $\overline{u}\,z$. A process can also receive a message on channel u with the blocking input action $u(v).P$. The parallel execution of two processes P and Q can be expressed as $P \mid Q$. Parallel processes can communicate by performing an input and an output action on the same channel; for example the process $\overline{u}\,z \mid u(v).P$ will perform an input and an output along channel u. Communication is described by the reaction $\overline{u}\,z \mid u(v).P \xrightarrow{\tau} P\{^z/_v\}$. Its effects are visible to the receiver as name substitution of the actual parameter z for the formal parameter v. The continuation P of the input process can be executed after the input on u has been received. In the polyadic pi calculus a message is a string of names $\tilde{v}$ instead of a single name v. The process $\nu u.P$ declare a local variable u with scope P. It is also possible to define a process that replicates itself: $!P$ is able to create an arbitrary number of copies of P. The pi calculus is summarized in Table 1: *labelled transitions* define the possible reactions of a process, contexts C are processes with holes filled by other processes, and represent environments. *Simulation* is a relation characterizing when two processes have the same behavior.

The general behavior of the protocol we propose is similar to that of the Business Transaction Protocol of OASIS. The expressed relation between transaction and the arbitrary nesting is also present in *Business Activities* (BA) of WS-Transactions.

A *two phase commit protocol* is used first to assemble the 'votes' of nested transactions (ie. whether or not they succeeded), and second to inform them all of the consensus decision. Additional features are provided for controlling which compensations are to be executed. We illustrate the features in Fig.1. where a holiday booking might succeed even if some sub-transactions (e.g. car rental) have failed; where, moreover, some sub-transactions (e.g. an Alitalia flight) might be cancelled even though the overall booking succeeds. The terminology used in [1] is that a *cohesor* needs only some of its children to succeed, while an *atom* requires them all to succeed. Cohesors are modelled here as entities able to flexibly specify the relation with their children. Atoms are a particular case of cohesor. We in fact consider a partition of all nested transactions into two groups: we require success from all of one *necessary* group, and we do not care about success of the other group. A transaction will report success only if all of its necessary children have succeeded. If a transaction fails then its sub transac-

Table 1. The asynchronous pi calculus

Terms P and contexts C in the asynchronous pi calculus are as follows. In $u(\widetilde{x})$ the names $\widetilde{x}$ are bound, as is x in $\nu x.P$. We identify terms up to alpha-renaming of bound names.

$$\begin{array}{lcl} P & ::= & 0 \;\mid\; \overline{u}\,\widetilde{x} \;\mid\; u(\widetilde{x}).P \;\mid\; P|P \;\mid\; \nu x.P \;\mid\; !P \\ C & ::= & \_ \;\mid\; u(\widetilde{x}).C \;\mid\; P|C \;\mid\; C|P \;\mid\; \nu x.C \;\mid\; !C \end{array}$$

Labelled transitions are as follows, where labels μ range over $u(\widetilde{x})$, $\nu\tilde{z}.\overline{u}\,\widetilde{x}$ and τ.

$$\overline{u}\,\widetilde{x} \xrightarrow{\overline{u}\,\widetilde{x}} 0 \quad (\textsc{out}) \qquad u(\widetilde{x}).P \xrightarrow{u(\widetilde{x})} P \quad (\textsc{in}) \qquad \frac{P\;|!P \xrightarrow{\mu} P'}{!P \xrightarrow{\mu} P'} \quad (\textsc{rep})$$

$$\frac{P \xrightarrow{\mu} P' \quad x \notin \mu}{\nu x.P \xrightarrow{\mu} \nu x.P'} \quad (\textsc{res}) \qquad \frac{P \xrightarrow{\nu\tilde{z}.\overline{u}\,\tilde{y}} P' \quad x \neq u, x \in \tilde{y}\backslash\tilde{z}}{\nu x.P \xrightarrow{\nu\tilde{z}x.\overline{u}\,\tilde{y}} P'} \quad (\textsc{open})$$

$$\frac{P \xrightarrow{\mu} P' \quad \mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\mu} P' \mid Q} \quad (\textsc{par}) \qquad \frac{P \xrightarrow{\nu\tilde{z}.\overline{u}\,\tilde{y}} P' \quad Q \xrightarrow{u(\tilde{x})} Q' \quad \tilde{z} \cap \mathrm{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\tau} \nu\tilde{z}.(P' \mid Q'\{\tilde{y}/\tilde{x}\})} \quad (\textsc{com})$$

Simulation is as follows. We write $\overset{\tau}{\Rightarrow}$ for $\xrightarrow{\tau}{}^{*}$, and $\overset{\mu}{\Rightarrow}$ for $\xrightarrow{\tau}{}^{*}\xrightarrow{\mu}\xrightarrow{\tau}{}^{*}$ when $\mu \neq \tau$, and $P \overset{\mu}{\Longrightarrow}$ for $\exists P' : P \overset{\mu}{\Longrightarrow} P'$. A symmetric relation $\mathcal{S}$ is a *weak ground simulation* if whenever $P\mathcal{S}Q$ then

– $P \xrightarrow{\mu} P'$ implies there exists Q' such that $Q \overset{\mu}{\Rightarrow} Q'$ and $P'\mathcal{S}Q'$.

Write $\lesssim$ for the largest ground simulation. S is a *weak ground bisimulation*, if both S and S^{-1} are weak ground simulations. Write $\approx$ for the largest ground bisimulation. We note some standard results:

$$P \approx Q \quad \text{implies} \quad \forall C : C[P] \approx C[Q] \qquad\qquad \nu x.x().P \approx 0$$

$$P|0 \approx P \qquad P|Q \approx Q|P \qquad P|(Q|R) \approx (P|Q)|R \qquad !P \approx P|!P$$

$$\nu x.\nu y.P \approx \nu y.\nu x.P \qquad \nu x.(P|Q) \approx P|\nu x.Q \text{ if } x \notin \mathrm{fn}(P)$$

$$\nu x.P \approx \nu x'.P\{^{x'}/_{x}\} \text{ if } x' \notin \mathrm{fn}(P)$$

Notation. We write $\widetilde{x}_{\mathcal{C}}$ for an arbitrary sequence $x_1, \ldots, x_n$ of the elements in set $\mathcal{C}$. We also use these syntactic sugars:

$$\begin{array}{rcll} x.P & = & x().P & \text{(empty input)} \\ \widetilde{x}.P & = & x_1.\ldots.x_n.P & \text{(sequence input)} \\ \nu\widetilde{x}.P & = & \nu x_1.\ldots.\nu x_n.P & \text{(sequence restriction)} \\ & & & \\ P \oplus Q & = & \nu c.(\overline{c}\,|c.P|c.Q),\ c \text{ fresh} & \text{(nondeterministic choice)} \\ & & & \\ x[P,Q] & = & \nu u,v.(\overline{x}\,u,v|u.P|v.Q),\ u,v \text{ fresh} & \text{(selection)} \\ \overline{x}\,\mathsf{left} & = & x(u,v).\overline{u} & \\ \overline{x}\,\mathsf{right} & = & x(u,v).\overline{v} & \end{array}$$

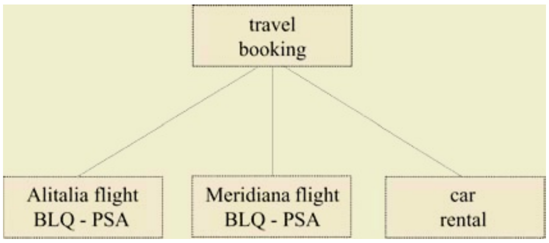


Fig. 1. A prototypical set of nested transactions. Each box represents a transactional web service, provided by different companies. There are several possible modes of failure propagation: *up-propagation*, where if Alitalia and Meridiana fail then we abort the car and the overall booking fails; *non-propagation*, where even if the car fails we can still proceed with the others; *down-specific-propagation*, where if one of Alitalia or Meridiana succeed then the other should be aborted; *down-propagation*, where the booking (the current job) is told to abort by some higher-up agent (not pictured) and so must abort all its children; *spontaneous-failure*, where the booking itself might decide to fail and so must abort its children.

tions fail (partial atomicity). If a transaction succeeds we also consider a second partition of the nested transactions: we will accept the success of the first group, and will abort the others. For instance, Fig.1. uses the following subsets:

If all in this subset succeeded...	*then accept these...*	*and undo these*
{Alitalia}	{Alitalia,car}	{Meridiana}
{Meridiana}	{Meridiana, car}	{Alitalia}

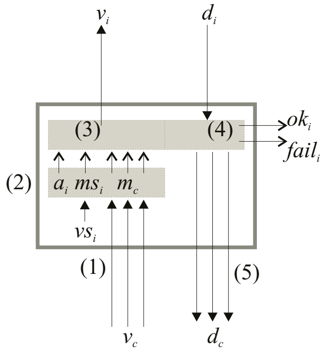
To simplify matters, the work in this paper considers only a single row of the table (ie. one necessary/unnecessary partition and one accept/reject partition), rather than multiple rows in each protocol specification. To handle multiple rows, something like Join patterns [16] might be used.

3 Design of Transaction Managers

In this section we implement (nested) transactions and their compensation-triggering. We implement them in the asynchronous pi calculus, using a generalization of the two phase commit implementation given by Berger and Honda [17].

We assume a set $\mathcal{I}$ of transactions ranged over by i. The tree-like hierarchy of these transactions is denoted by a relation par : $\mathcal{I} \mapsto \mathcal{I}$ which indicates the immediate parent of a transaction; writing $\mathrm{par}^n(i)$ for n applications of the pair function, we assume that if $i = \mathrm{par}^m(j)$ then do not exists n such that $j = \mathrm{par}^n(i)$. Define the set of i's children $C(i) = \{j : \mathrm{par}(j) = i\}$. As discussed in the introduction, we consider only a single 'necessary' partition of $C(i)$ into $N(i), U(i)$ – with the meaning that success of all $N(i)$ is necessary for i to succeed, while $U(i)$ are unnecessary. We therefore consider just a single consequent partition of $C(i)$ into $A(i), R(i)$ – where all of $A(i)$ are accepted, while all of $R(i)$ are rejected (undone).

We now describe the operation of each transaction block. We illustrate with transaction i, which has children $\widetilde{c}$.



(1) This transaction i itself makes a non-deterministic ‘self’ vote vs_i. Also, all of the children c make their votes v_c. All these votes are made using left/right notation (Table 1). **(2)** Each vote is transformed into an ‘internal message’ m. The purpose of this translation is to separate necessary child votes $N(i)$ from unnecessary votes $U(i)$. Each internal message m_c means that the child either voted success (left), or it was unnecessary. But if a child should vote failure (right) and was necessary, it will make an ‘abort’ signal a_i instead. **(3)** If all internal messages ms_i/m_c arrive, then the transaction i as a whole can succeed, and so indicates success (left) to its parent over the channel v_i. But if even one abort message a_i was received, then the transaction as a whole fails, and so it indicates failure (right). **(4)** Eventually the parent p will know whether to accept i, or to abort/undo it. This decision is communicated to i via the ‘decision’ channel d_i, and so determines i’s final state. The transaction i can indicate its final state via the messages $ok_i/abort_i$. **(5)** Finally, the decision is propagated down to all the children c. The accepted children, those in $A(i)$, will be told the same decision as i received. The rejected children, those in $R(i)$, will be told to abort/undo regardless. The code T_i for transaction i plus all its descendants, is as follows.

$$T_i = \nu a_i, ms_i, vs_i, \widetilde{m}_{C(i)}, \widetilde{v}_{C(i)}, \widetilde{d}_{C(i)}. \quad (T_i.\mathrm{sv} \mid T_i.\mathrm{m} \mid T_i.\mathrm{col} \mid \prod_{c \in C(i)} T_c) \qquad \text{(transaction)}$$

$$T_i.\mathrm{sv} = \overline{vs}_i left \oplus \overline{vs}_i right \qquad \text{(self-vote)}$$

$$T_i.\mathrm{m} = vs_i[\overline{ms}_i, \overline{a}_i] \mid \prod_{c \in N(i)} v_c[\overline{m}_c, \overline{a}_i] \mid \prod_{c \in U(i)} v_c[\overline{m}_c, \overline{m}_c] \qquad \text{(internals)}$$

$$T_i.\mathrm{col} = a_i.(\overline{v}_i right \mid T_i.\mathrm{fail}) \mid \widetilde{m}_{C(i)}.ms_i.(\overline{v}_i left \mid d_i[T_i.\mathrm{ok}, T_i.\mathrm{fail}]) \qquad \text{(collate votes)}$$

$$T_i.\mathrm{ok} = \overline{ok}_i \mid \prod_{c \in A(i)} \overline{d}_c left \mid \prod_{c \in R(i)} \overline{d}_c right \qquad \text{(ok)}$$

$$T_i.\mathrm{fail} = \overline{abort}_i \mid \prod_{c \in C(i)} \overline{d}_c right \qquad \text{(fail)}$$

To explain the code, the *transaction* T_i consists of three parts: T_i.sv generates its self-vote, T_i.m fulfills step (2) by converting votes into internal messages, and T_i.col collates votes and receives the final decision, for steps (3–5). We have also included all the children transactions T_c, since they refer to the local channels $\widetilde{v}_{C(i)}$ and $\widetilde{d}_{C(i)}$.

The *self-vote* T_i.sv makes a non-deterministic choice (using $\oplus$) to become, at runtime, a vote for success or failure.

The *internals* T_i.m convert all the votes into internal messages according to whether the vote came from a needed component $N(i)$ or an unnecessary one $U(i)$. We count the self-vote as necessary. An internal message m_c is generated if the child c voted success, or if the child c was unnecessary. An internal abort message a_i is generated otherwise (i.e. a necessary child voted for failure).

The *collator* T_i.col will either receive all the internal messages m_c/ms_i, or will receive at least one internal abort a_i. The abort signifies that a necessary part failed. If this happens, then the component i signals a failure to its parent on channel v_i, and proceeds with T_i.fail to tell abort its children. But if all internal messages were received, then it tells its parent about its success, and awaits the parent's final verdict.

The *ok/fail* processes T_i.ok and T_i.fail indicate the final state of this transaction, using the global channels $\overline{ok}_i$ and $\overline{abort}_i$. In the case of OK, the accepted children $A(i)$ are told of the positive verdict, while the rejected children $R(i)$ are told to fail. In the case of FAIL, all children are told to fail.

Let us recall the five modes of propagation identified in Fig.1. and explain how they are reflected in the code. Let us denote the Alitalia transaction with i_a, Meridiana with i_m, car rental with i_c and travel booking with i. As an example we consider the following row:

If all in this subset succeeded...	*then accept these...*	*and undo these*
{Alitalia}	{Alitalia,car}	{Meridiana}

Up-propagation is achieved by enclosing i_a in the set $N(i)$ so that if T_i.m receives the failure vote $\overline{v}_{i_a}$ it eventually fails. It fails by sending a message $\overline{a}_i$ that unblocks the abort branch of T_i.col. *Non-proragation* is achieved by enclosing i_c in the set $U(i)$ so that T_i.m reacts to both success and failure messages $\overline{m}_{i_c}$. *Down-specific proragation* is achieved by enclosing i_m in the set $R(i)$ so that T_i.ok communicates, in any case, a failure decision to i_m. *Down proragation* is implemented by the message $\overline{d}_i$ that in case of local success of i notifies the upper outcome.

Finally, we collect the overall tree of transactions in a test harness H. We suppose the root of the tree is transaction i:

$$H = \nu v_i, d_i. \left(T_i \mid v_i[\overline{d}_i left, \overline{d}_i right] \right).$$

This harness merely executes the root transaction T_i, waits for its overall vote v_i, and immediately sends back the vote as the decision if vote was success. If vote was fail there is no need of decision communication: the child already had its outcome without waiting any signal.

The following lemma describes the observable behavior a of generic transactions. T_i takes a local decision on the basis of the votes of its children (if there are any) and of its non deterministic self-vote. In any case T_i communicates its vote to the parent. If it locally failed it terminates with failure soon. If it did not locally fail it waits for the global decision of the parent and its final outcome depends from it.

Lemma 1. *If $C(i) = \emptyset$, then $T_i \approx (\overline{v}_i left \mid d_i[\overline{ok}_i, \overline{abort}_i]) \oplus (\overline{v}_i right \mid \overline{abort}_i)$.*

Proof sketch. When $C(i) = \emptyset$, then also $N(i) = U(i) = A(i) = R(i) = \emptyset$. Hence T_i simplifies to just

$$\begin{aligned} T_i =& \nu a_i, ms_i, vs_i.(\, (\overline{vs_i}\, left \oplus \overline{vs_i}\, rightto) \mid vs_i[\overline{ms_i}\,, \overline{a_i}\,] \\ & \mid a_i.(\overline{v_i}\, right \mid \overline{abort}_i) \mid ms_i.(\overline{v_i}\, left \mid d_i[\overline{ok_i}\,, \overline{abort}_i])\,). \end{aligned}$$

Observe that the only action T_i can make is a τ move, choosing whether vs_i votes left or right. This is reflected by the right hand side.

4 Transaction Properties

In this section we prove some properties of the protocol: *Durability*, *Eventuality* and *Local Atomicity*. Durability means that each node reaches no more than one outcome and, in general, that the only observable behavior of the protocol is the set of outcome notifications. Eventuality implies the absence of deadlock in the protocol: an outcome is achieved in every node of the tree. Finally we consider Local Atomicity. Normally, atomicity is the property that either every transaction succeeds or every transaction fails. We have seen that this is too strict for business transactions. Instead, local atomicity is just the property that if one transaction fails, then all its children fail. Let us start by defining a transaction's descendants set.

Definition 2 (Descendants). *Define $D(i) = \{j : \exists n. i = \mathrm{par}^n(j)\}$.*

The precise pattern of the 'mountains' (Fig.2.) is determined by the compile-time choice of which failures propagate, ie. by the partitions of $D(i)$, $N(i)/U(i)$ and $A(i)/R(i)$, and also by the run-time non-deterministic self-vote made by each transaction. We start with the proposition that, after the transaction has finished executing, it ends up in a state where every node has made a single choice (either $\overline{ok}_i$ or $\overline{abort}_i$), such that the set of all nodes respects local atomicity.

Before starting the lemmas we remark upon conventions. Recall the syntactic sugar for selection (Table 1):

$$\begin{aligned} x[P,Q] &= \nu u, v.(\overline{x}\, u, v \mid u.P \mid v.Q), \text{with } u, v \text{ fresh} && \text{(selection)} \\ \overline{x}\,\mathsf{left} &= x(u,v).\overline{u} \\ \overline{x}\,\mathsf{right} &= x(u,v).\overline{v} \end{aligned}$$

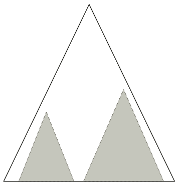


Fig. 2. Local atomicity may be pictured as 'mountains', where the shaded mountains represents those nodes, in the transactions tree, that have failed.

We will use shorthand labels $x(left), x(right), \overline{x}\,left, \overline{x}\,right$, with the following transitions:

$$x[P,Q] \overset{x(left)}{\Longrightarrow} P \qquad x[P,Q] \overset{x(right)}{\Longrightarrow} Q$$

$$\overline{x}\,left \overset{\overline{x}\,left}{\Longrightarrow} 0 \qquad \overline{x}\,right \overset{\overline{x}\,right}{\Longrightarrow} 0$$

$$\frac{P \overset{\overline{x}\,left}{\Longrightarrow} P' \quad Q \overset{x(left)}{\Longrightarrow} Q'}{P \mid Q \overset{\tau}{\Longrightarrow} P' \mid Q'} \qquad \frac{P \overset{\overline{x}\,right}{\Longrightarrow} P' \quad Q \overset{x(right)}{\Longrightarrow} Q'}{P \mid Q \overset{\tau}{\Longrightarrow} P' \mid Q'}$$

and also equivalent versions of (RES) and (PAR). These rules are satisfactory abstractions of the actual selection transitions, so long as the process in question only ever uses selection channels appropriately (e.g. there is no $x[P,Q] \mid x(u,v).(\overline{u} \mid \overline{v}))$. In some cases we want to refer to generic actions, i.e. votes and decisions, indifferently from their specific types: $\overline{v}_i$ stands for a generic vote from i ($\overline{v}_i left$ or $\overline{v}_i right$ arbitrarily), $\overline{d}_i$ stands for an arbitrary decision ($\overline{d}_i left$ or $\overline{d}_i right$). We will also refer to generic outcome notifications ($\overline{abort}_i$ or $\overline{ok}_i$) with $\overline{ouctome}_i$.

Durability

We now prove durability: the observable behavior is never anything other than a single outcome notification ($\overline{ok}_i/\overline{abort}_i$) for each node. The property is proved in Theorem 5; we present some auxiliary lemmas first.

Lemma 3.

1. $\nu a_i, ms_i, \widetilde{m}_{C(i)}.(T_i.m \mid a_i.P \mid \widetilde{m}_{C(i)}.ms_i.Q) \lesssim vs_i[0,0] \mid P \oplus Q \mid \prod_{c \in C(i)} v_c[0,0]$.
2. $P \oplus d_i[Q,P] \lesssim P \oplus (d_i[0,0] \mid P \oplus Q)$.
3. $(P_1 \mid P_2) \oplus (Q_1 \mid Q_2) \lesssim (P_1 \oplus Q_1) \mid (P_2 \oplus Q_2)$.
4. $P_1 \oplus P_2 \approx P_1 \oplus (P_2 \oplus P_1)$.

Proposition 4. $\nu v_i, d_i.T_i \lesssim \prod_{j \in D(i)} (\overline{abort}_j \oplus \overline{ok}_j)$.

Proof. By induction on the depth of the tree.
Base Case. $C(i) = \emptyset$. By Lemma 1, $T_i \approx (\overline{v}_i left | d_i[\overline{ok}_i, \overline{abort}_i]) \oplus (\overline{v}_i right | \overline{abort}_i)$. Applying now Lemma 3 (case 2, 1 and 3) to the right hand term

$$T_i \lesssim (\overline{ok}_i \oplus \overline{abort}_i) \mid (\overline{v}_i left \oplus \overline{v}_i left) \mid d_i[0,0].$$

Trivially $\nu v_i, d_i.T_i \lesssim (\overline{abort}_i \oplus \overline{ok}_i)$.

Inductive Case. let us consider a generic node T_i. We have for inductive hypothesis that $\forall c \in C(i), \nu v_c, d_c.T_c \lesssim \prod_{j\in D(c)}(\overline{abort}_j \oplus \overline{ok}_j)$. It is straightforward that

$$\nu\widetilde{v}_{C(i)}, \widetilde{d}_{C(i)}. \prod_{c\in C(i)} T_c \lesssim A \text{ where } A = \prod_{c\in C(i)} \prod_{j\in D(c)} (\overline{abort}_j \oplus \overline{ok}_j).$$

By the standard results properties of $\approx$ (Table 1) we have the following:

$$T_i \approx \quad \nu vs_i, \widetilde{v}_{C(i)}, \widetilde{d}_{C(i)}.(\prod_{c\in C(i)} T_c \mid T_i.\text{sv} \mid \nu a_i, ms_i, \widetilde{m}_{C(i)}.(T_i.\text{m} \mid T_i.\text{col}))$$

$$\lesssim \quad \nu vs_i, \widetilde{v}_{C(i)}, \widetilde{d}_{C(i)}.(A \mid T_i.\text{sv} \mid vs_i[0,0] \mid \prod_{c\in C(i)} v_c[0,0] \mid P \oplus Q)$$

(Lemma 3.1)

$$\lesssim \quad \nu\widetilde{d}_{C(i)}.((P \oplus Q) \mid \nu\widetilde{v}_{C(i)}.(\prod_{c\in C(i)} v_c[0,0]) \mid \nu vs_i.(vs_i[0,0] \mid T_i.\text{sv}) \mid A.$$

(structural)

Trivially $\nu vs_i.(vs_i[0,0] \mid T_i.\text{sv}) \approx 0$ and $\nu\widetilde{v}_{C(i)}.(\prod_{c\in C(i)} v_c[0,0]) \approx 0$ so

$$T_i \quad \lesssim \quad \nu\widetilde{d}_{C(i)}.((P \oplus Q) \mid A)$$

where $P = \overline{v}_i right \mid T_i.\text{fail}$ and $Q = \overline{v}_i left | d_i[T_i.\text{ok}, T_i.\text{fail}])$. By Lemma 3.2,

$$T_i \lesssim \quad \nu\widetilde{d}_{C(i)}.((T_i.\text{fail} \oplus d_i[T_i.\text{ok}, T_i.\text{fail}]) \mid (\overline{v}_i right \oplus \overline{v}_i left) \mid A)$$

$$\approx \nu\widetilde{d}_{C(i)}.((T_i.\text{fail} \oplus (d_i[0,0] \mid (T_i.\text{ok} \oplus T_i.\text{fail}))) \mid (\overline{v}_i right \oplus \overline{v}_i left) \mid A)$$

(Lemma 3.1)

$$\approx \nu\widetilde{d}_{C(i)}.((\overline{abort}_i \oplus (\overline{ok}_i \oplus \overline{abort}_i)) \mid R \mid d_i[0,0] \mid (\overline{v}_i right \oplus \overline{v}_i left) \mid A)$$

(Lemma 3.2)

where R is obtained extracting the decision propagation from T_i.ok and T_i.fail

$$R = \prod_{c\in C(i)} \overline{d}_c right \oplus ((\prod_{c\in A(i)} \overline{d}_c left \mid \prod_{c\in R(i)} \overline{d}_c right) \oplus \prod_{c\in C(i)} \overline{d}_c right).$$

$$\approx \quad (\overline{abort}_i \oplus (\overline{ok}_i \oplus \overline{abort}_i)) \mid \widetilde{d}_{C(i)}.R \mid d_i[0,0] \mid (\overline{v}_i right \oplus \overline{v}_i left) \mid A$$

(structural)

$$\approx \quad (\overline{abort}_i \oplus (\overline{ok}_i \oplus \overline{abort}_i)) \mid d_i[0,0] \mid (\overline{v}_i right \oplus \overline{v}_i left) \mid A$$

(structural)

$$\lesssim \quad (\overline{abort}_i \oplus \overline{ok}_i) \mid d_i[0,0] \mid (\overline{v}_i right \oplus \overline{v}_i left) \mid A.$$

(Lemma 3.3)

Note that $\nu v_i.(\overline{v}_i right \oplus \overline{v}_i left) \approx \mathsf{0}$ and $\nu d_i.d_i[\mathsf{0},\mathsf{0}] \approx \mathsf{0}$, so (structural)

$$\nu v_i, d_i T_i \lesssim \prod_{j \in D(i)} (\overline{abort}_j \oplus \overline{ok}_j).$$

Corollary 5 (Durability). $H \lesssim \prod_{i \in \mathcal{I}} (\overline{abort}_j \oplus \overline{ok}_j)$.

Eventuality

We prove that any node of the tree can always notify an outcome (none of the nodes deadlocks). Lemma 6 proves that each transaction can eventually vote for each possible computation. Lemma 7 provides that, depending on the vote of a node, we can always get a decision from the parent that unblocks one of the final processes T_i.ok or T_i.fail. Then we prove (Lemma 8) that if a node votes and its provided with a decision then it and all the subtree is able to have an outcome. This leads directly to Corollary 9 that deals with the observable behavior of H. The following lemma is that a transaction can always eventually vote, no matter what sequence of internal moves it has already made.

Lemma 6. *If $T_i \overset{\tau}{\Longrightarrow} T_i'$ then $T_i' \overset{\overline{v}_i left}{\Longrightarrow}$ or $T_i' \overset{\overline{v}_i right}{\Longrightarrow}$.*

Henceforth we use the shorthand $\tilde{z} = a_i, ms_i, vs_i, \widetilde{m}_{C(i)}, \widetilde{v}_{C(i)}, \widetilde{d}_{C(i)}$ to refer to the scope of a node T_i.

Lemma 7.

1. *If $T_i \overset{\overline{v}_i left}{\Longrightarrow} T_i'$ then $T_i' \overset{d_i(left)}{\Longrightarrow} \nu\tilde{z}.(T_i.ok \mid P)$ for some P and $T_i' \overset{d_i right}{\Longrightarrow} \nu\tilde{z}.(T_i.fail \mid Q)$ for some Q,*
2. *If $T_i \overset{\overline{v}_i left}{\Longrightarrow} T_i'$ then $T_i' \overset{\tau}{\Longrightarrow} \nu\tilde{z}.(T_i.fail \mid P)$ for some P.*

Proof sketch. 1. If T_i was able to perform a $\overline{v}_i left$ transition it has previously unblocked the successful branch of T_i.col that is $\overline{v}_i left \mid d_i[T_i.\text{ok}, T_i.\text{fail}]$. After the $\overline{v}_i left$ transition it will become

$$\nu\tilde{z}.(a_i.(\overline{v}_i right \mid T_i.\text{fail}) \mid d_i[T_i.\text{ok}, T_i.\text{fail}] \mid \prod_{c \in C(i)} T_c').$$

The only possible transitions are the following:

$$\xrightarrow{d_i(left)} \nu\tilde{z}.(a_i.(\overline{v}_i right \mid T_i.\text{fail}) \mid T_i.\text{ok} \mid \prod_{c \in C(i)} T_c'), \text{ or}$$

$$\xrightarrow{d_i(right)} \nu\tilde{z}.(a_i.(\overline{v}_i right \mid T_i.\text{fail}) \mid T_i.\text{fail} \mid \prod_{c \in C(i)} T_c').$$

2. If T_i is able to perform a $\overline{v}_i right$ action it has already triggered the failing action of T_i.com. After sending the output $\overline{v}_i right$ the process of the node i is as follows:

$$\nu\tilde{z}.(T_i.\text{fail} \mid \widetilde{m}_C.ms_i.(\overline{v}_i left \mid d_i[T_i.\text{ok}, T_i.\text{fail}]) \mid \prod_{c\in C(i)} T_c').$$

   Note: by hypothesis, only τ moves have been performed hence T_i.fail has not reacted.

Lemma 8. *If $T_i' : T_i \overset{\overline{v}_i left}{\Longrightarrow} T_i'$ or $T_i \overset{\overline{v}_i right}{\Longrightarrow} T_i'$ then for every $j \in \{i\} \cup D(i)$, we have $T_i' \mid \overline{d}_i \overset{\overline{ok}_j}{\Longrightarrow}$ or $T_i' \mid \overline{d}_i \overset{\overline{abort}_j}{\Longrightarrow}$.*

Proof. Let us reason by induction on the depth of the level of i.
Base Case. $C(i) = \emptyset$. By Lemma 6, $T_i \xrightarrow{\overline{v}_i} T_i'$. Thus by Lemma 7 $T_i' \mid \overline{d}_i \overset{\overline{outcome}_i}{\Longrightarrow}$.
Inductive Case. By Lemma 6 we have $T_i \xrightarrow{\overline{v}_i} T_i'$. By Lemma 7 we have $T_i' \mid \overline{d}_i \overset{\tau}{\Longrightarrow} \nu\tilde{z}.(T_i.\text{ok} \mid P)$ for some P or $T_i' \mid \overline{d}_i \overset{\tau}{\Longrightarrow} \nu\tilde{z}.(T_i.\text{fail} \mid P)$ for some P. Recall the definition of T_i.ok and $T_i.abort$:

$$\begin{aligned} T_i.\text{ok} &= \overline{ok}_i \mid \prod_{c\in A(i)} \overline{d}_c left \mid \prod_{c\in R(i)} \overline{d}_c right \\ T_i.\text{fail} &= \overline{abort}_i \mid \prod_{c\in C(i)} \overline{d}_c right \end{aligned}$$

In both cases it is possible, from $\nu\tilde{z}.(T_i.\text{ok} \mid P)$, to perform the following actions:

– $\xrightarrow{\overline{d}_c}$ such that by inductive hypothesis $\forall j \in D(c) \cup \{c\}$, $T_c' \mid \overline{d}_c \overset{\overline{outcome}_j}{\Longrightarrow}$,
– $\xrightarrow{\overline{outcome}_i}$.

It holds so that for any T_i' such that $T_i \overset{\overline{v}_i}{\Longrightarrow} T_i'$ then for every $j \in \{i\} \cup D(i)$, $T_i' \mid \overline{d}_i \overset{\overline{outcome}_j}{\Longrightarrow}$.

Corollary 9 (Eventuality). *For every H' such that $H \overset{\tau}{\Longrightarrow} H'$ then, for every $j \in \mathcal{I}$, where $H' \overset{\overline{ok}_i}{\Longrightarrow}$ or $H' \overset{\overline{abort}_i}{\Longrightarrow}$.*

Local Atomicity

To prove Local Atomicity we simplify (Lemma 10) the behavior of T_i by considering its state after the it voted. Recall that by Lemma 6, each node eventually votes. Then we show (Lemma 11) that if a node i receives a failure decision or votes failure itself then none of the nodes in the subtree of i will ever notify a successful outcome. Finally we show that any node abort just if it received a failure notification or votes failure itself and generalize the property to the whole protocol H.

Lemma 10.

1. *If* $T_i \overset{\overline{v}_i left}{\Longrightarrow} T_i'$ *then* $T_i' \approx \nu\widetilde{d}_{C(i)}.(d_i[T_i.ok, T_i.fail] \mid \prod_{c\in C(i)} T_c')$ *with* $T_c \overset{\overline{v}_c}{\Longrightarrow} T_c'$.
2. *If* $T_i \overset{\overline{v}_i right}{\Longrightarrow} T_i'$ *then* $T_i' \approx \nu\widetilde{d}_{C(i)}.(T_i.fail \mid \prod_{c\in C(i)} T_c')$ *with* $T_c \overset{\overline{v}_c}{\Longrightarrow} T_c'$ *or* $T_c \Rightarrow T_c'$.

Lemma 11.

1. *If* $T_i \overset{\overline{v}_i left}{\Longrightarrow} T_i'$ *then* $\nexists j \in \{i\} \cup D(i)$ *such that* $T_i' \mid \overline{d}_i right \overset{\overline{ok}_j}{\Longrightarrow}$,
2. *if* $T_i \overset{\overline{v}_i right}{\Longrightarrow} T_i'$ *then* $\nexists j \in \{i\} \cup D(i)$ *such that* $T_i' \overset{\overline{ok}_j}{\Longrightarrow}$.

Proof. For induction on the depth of the tree.
Base Case. the tree is composed by node T_i with $C(i) = \emptyset$. From Lemma 1 $T_i \approx Ts_i$ with $Ts_i = (\overline{v}_i left | d_i[\overline{ok}_i, \overline{abort}_i]) \oplus (\overline{v}_i right | \overline{abort}_i)$. The only step that Ts_i can perform is a τ action corresponding to the choice of one of the two branches.

1. If the left branch is chose then the only possible sequence of steps is: $Ts_i \xrightarrow{\tau} \overline{v}_i left | d_i[\overline{ok}_i, \overline{abort}_i] \overset{\overline{v}_i left}{\Longrightarrow} d_i[\overline{ok}_i, \overline{abort}_i]$. Hence $d_i[\overline{ok}_i, \overline{abort}_i] \mid \overline{d}_i right \overset{\tau}{\Longrightarrow} \overline{abort}_i \overset{\overline{abort}_i}{\Longrightarrow}$.
2. If the right branch is chose then the only possible sequence of steps is: $Ts_i \xrightarrow{\tau} \overline{v}_i right | \overline{abort}_i \overset{\overline{v}_i right}{\Longrightarrow} \overline{abort}_i \overset{\overline{abort}_i}{\Longrightarrow}$.

Inductive Case. we have that $T_i \overset{\overline{v}_i}{\Longrightarrow} T_i'$. Depending on the vote type we have two cases:

1. If $T_i \overset{\overline{v}_i left}{\Longrightarrow} T_i'$ then by Lemma 10 $T_i' \approx \nu\widetilde{d}_{C(i)}.(d_i[T_i.\mathrm{ok}, T_i.\mathrm{fail}] \mid \prod_{c\in C(i)} T_c')$. Hence
$$\nu d_i.(T_i' \mid \overline{d}_i right) \approx T_i.\mathrm{fail} \mid \prod_{c\in C(i)} T_c').$$
Here i will surely fail (and just fail for durability (Theorem 5)) and will also provide a $\overline{d}_c right$ decision $\forall c \in C(i)$ that for inductive hypothesis grant that $\nexists j \in \{c\} \cup D(c)$ such that $T_c' \mid \overline{d}_i right \overset{\overline{ok}_j}{\Longrightarrow}$.
2. If $T_i \overset{\overline{v}_i right}{\Longrightarrow} T_i'$ then for Lemma 10 $T_i' \approx \nu\widetilde{d}_{C(i)}.(T_i.\mathrm{fail} \mid \prod_{c\in C(i)})$. The case is analogue to the previous one here.

Theorem 12 (Local Atomicity). *If* $H \overset{\overline{abort}_i}{\Longrightarrow} H'$ *then* $\nexists j \in D(i)$ *such that* $H' \overset{\overline{ok}_j}{\Longrightarrow}$.

Proof. If $H \overset{\overline{abort}_i}{\Longrightarrow} H'$ then i must have failed for one of the following reasons:

– i voted $\bar{v}_i right$, the result follows from Lemma 11.2,
– i voted $\bar{v}_i left$ and received a failure decision from the parent $\bar{d}_i right$. The result follows from Lemma 11.1.

5 Conclusions

We discussed a possible behavior for long running transactions in a context of hierarchical relations with other transactions, represented by an arbitrarily deep tree. The exercise had the aim to clarify two principal aspects.

The first is the role of *cohesors* and *atoms* in the protocol, their behavior and their relation. We proposed a flexible approach for describing the relation between votes of a sub-transaction and parent outcome type, and again between parent outcome and children outcome. Atoms can be modelled here as particular cases of cohesors. This flexible behavior is present in also in BTP and WS-Transactions. The paper provides an implementation with the pi calculus.

The second aspect discussed in this paper is the mechanism of compensation triggering. Compensations are a straightforward addition to the current work: each failure notification $\overline{abort_i}$ is associated to the execution of the compensation of transaction i. Transactions are thought as independent entities, maybe from different companies, connected by a superior-inferior (caller-provider) links. Those links create a hierarchical structure of arbitrary depth. Our mechanism coordinates the triggering of compensations: when a node i fails the protocol creates the global compensation process by composing the local compensations of all the nodes in the subtree of i.

We proved that each transaction has no more than one outcome (Durability) and that the protocol does not deadlock (Eventuality). We also proved that if one node fails then its entire subtree will also fail (Local Atomicity).

Other aspects have yet to be considered. It would be desirable to allow an explicit representation of the choice of sets $N(i), A(i), U(i), R(i)$ at run time, according to the computation feedback. This aspect is indirectly managed (it could be simulated with Join patterns). Other aspects are the introduction of concepts like localities and unreliability in communication between remote transactions. Managing these would probably lead to the introduction of timers in order to avoid deadlock pathologies.

Acknowledgement

The technical part of this paper owes much to discussions with Lucian Wischik, and the general ideas on transaction behavior also owe much to Greg Meredith. I thank Paolo Ciancarini, Cosimo Laneve, Gianluigi Zavattaro and the anonymous referees for their useful comments.

References

1. S. Dalal, S. Temel, M. Little, M. Potts and J. Webber. Coordinating Business Transactions on the Web. IEEE Internet Computing, January-February 2003.
2. J.J. Dubray. A novel approach for modeling business process definitions. [http://www.ebpml.org/ebpml2.2.doc].

3. S. Thatte. XLANG: Web Services for Business Process Design. [http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm], Microsoft Corporation, 2001.
4. F. Leymann. Web Services Flow Language (WSFL 1.0). [http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf], Member IBM Academy of Technology, IBM Software Group, 2001.
5. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte and S. Weerawarana. Business Process Execution Language for Web Services (BPEL4WS 1.0). [http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/], 2002.
6. W3C Choreography Group. Web Services Choreography Requirements 1.0. [http://www.w3.org/TR/ws-chor-reqs/], 2003.
7. H. Garcia-Molina and K. Salem. Sagas. In Proc. of SIGMOND International Conference on Management of Data, pp. 249-259, 1987.
8. H. Garcia-Molina, G. Gawlick, J. Klein, K. Kleissner and K. Salem. Salem. Modelling Long Running Activities as Nested Sagas. IEEE Bulletin of Technical Committee on Data Engeneering, 14(1), 1991.
9. J. Roberts and K. Srinivasan. Tentative Hold Protocol Part 1: White paper. W3C Note 28 November 2001. [http://www.w3.org/TR/tenthold-1/]
10. F. Cabrera, G. Copeland, B. Cox, T. Freund, J. Klein, T. Storey and S. Thatte. Web Services Transaction (WS-Transaction). [http://www-106.ibm.com/developerworks/webservices/library/ws-transpec/], 2002.
11. R. Bruni, C. Laneve and U. Montanari. Orchestrating Transactions in Join Calculus. In CONCUR'02, LNCS 2421, pp. 321-337.
12. L. Bocchi, C. Laneve and G. Zavattaro. A Calculus for Long Running Transactions. In FMOODS'03, LNCS 2884, pp. 124-138.
13. P. K. Chrysantys and K. Ramamritham. ACTA: a framework for specifying and reasoning about transaction structure and behavior. In SIGMOND International Conference on Management of Data, pp. 194-203, 1990.
14. L. Lamport. The Temporal Logic of Actions. ACM TOPLAS, 16(3), pp. 872-923, 1994.
15. R. Milner. Communicating and Mobile Systems: the Pi-calculus. Cambridge University Press, 1999.
16. F. Le Fessant and L. Maranget. Compiling Join-Patterns. Electronic Notes in Theoretical Computer Science 16(3), 2000.
17. M. Berger and K. Honda. The Two-Phase Commitment Protocol in an Extended Pi-calculus. Electronic Notes in Theoretical Computer Science 39(1), 2003.

DaGen: A Tool for Automatic Translation from DAML-S to High-Level Petri Nets

Daniel Moldt and Jan Ortmann

University of Hamburg, Computer Science Department
Vogt-Kölln-Str. 30, D-22527 Hamburg
{moldt,ortmann}@informatik.uni-hamburg.de

Abstract. The Internet provides possibilities for distributed execution of business processes and Web Services. This caused the emergence of a variety of Web Services that might be composed to accomplish tasks. To efficiently compose these tasks a simple workflow description no longer suffices. We therefore suggest a description based on high-level Petri nets called reference nets, allowing for the consideration of pre- and post-conditions of services offered on the Internet. We demonstrate how DAML-S models can be automatically translated into high-level nets and thus can be directly executed in such contexts, including Petri net based MAS.

Keywords: high-level Petri nets, nets within nets, reference nets, Renew, workflow, web service, business process, DAML-S, process ontology

1 Introduction

The modelling of workflows and processes with Petri nets has been thoroughly investigated (see [2, 16, 1]). Petri nets offer a clear semantics and a rather intuitive way to process modelling. This makes them both easy to handle for humans and unambiguously processable by machines. Based on Renew[1] (Reference Net Workshop, see [13]), a Petri net simulator, written in Java, several approaches for the modelling of workflows with Petri nets have been made (see [7, 16]). We adopt those approaches and transfer them to the area of Web Services. The composition of Web Services is strongly related to the modelling of workflows.

Renew allows for the execution of Java code and is therefore capable of not only graphically describing a composition of Web Services but also of executing the described composition. Since process descriptions in DAML-S use XML, they are hard to read without support. Additionally DAML-S allows for concept hierarchies, so that a sub-concept might inherit properties from a super-concept. This makes it rather difficult to grasp the meaning of a DAML-S description. Based on Renew, we developed a tool to read DAML-S descriptions, show them as Petri net (reference net) and execute them from within the Renew simulator, following the ideas introduced in [14], that showed that DAML-S descriptions can be modelled as Petri nets.

[1] Renew is freely available from http://www.renew.de/

2 Conceptual Background

This tool heavily relies upon the Renew simulator. Renew is a Petri net simulator that allows for the graphical drawing and for the execution of reference nets (see [9, 12, 11, 10]). *Reference nets* represent an extension of the Coloured Petri net (CPN) formalism (see [8]) adding both the concept of nets within nets introduced by [15] and the concept of synchronous channels (see [5]). Additionally they provide means to execute Java code directly from within the Petri net through inscriptions on net elements. Furthermore, Renew has a plug-in mechanism, so that it can easily be extended by new functionality. Furthermore new net classes can be created, loaded as net instances and can therefore be executed.

Having started the Renew-application, two windows appear: a toolbar window and a drawing window. The former allows to pick a drawing tool which can than be used in the latter window. By picking places, transitions, arcs and inscriptions, the user can easily draw a Petri net. These Petri nets can than be executed, which will then be displayed in an extra window, so that the user can follow the execution of the net. Figure 1 shows the Renew-simulator with both windows mentioned above. Here a synchronous channel and execution of Java code (via the action inscription) is modelled. Via synchronous channels different net instances can communicate.

The Renew simulator fully supports the execution of reference nets. `action` inscriptions make the simulator execute the Java code following the `action` keyword. This way arbitrary code can be executed.

DAML-S[2] is a based on DAML+OIL[3] and provides the framework for a process ontology to describe Web Services. Just like DAML+OIL, DAML-S is closely related to description logics. It therefore allows for the discovery of Web Services through inference. A general category of Web Services can be investigated and the services best suited for a given task can be picked. Besides this categorisation of Web Services, DAMLS-S also provides information about the input and the output parameters as well as the preconditions and the effects of a service (IOPE).

3 Tool

The tool introduced here is a plug-in for Renew. It allows to read a DAML-S description, which will than be displayed as a set of Petri nets. The Petri nets generated allow different views on the process, such as the control flow, the data and the operations. These views were partly adopted from the views on workflows discussed in [3]. The different nets, that have been generated can be executed by the simulator. To exchange data, they communicate with each other through synchronous channels. In order to get a quick overview, the user is able to merely look at the control flow of a process and in order to get a deeper insight it is possible to consider each process separately along with its IOPE.

[2] http://www.daml.org/services/
[3] http://www.daml.org/language/

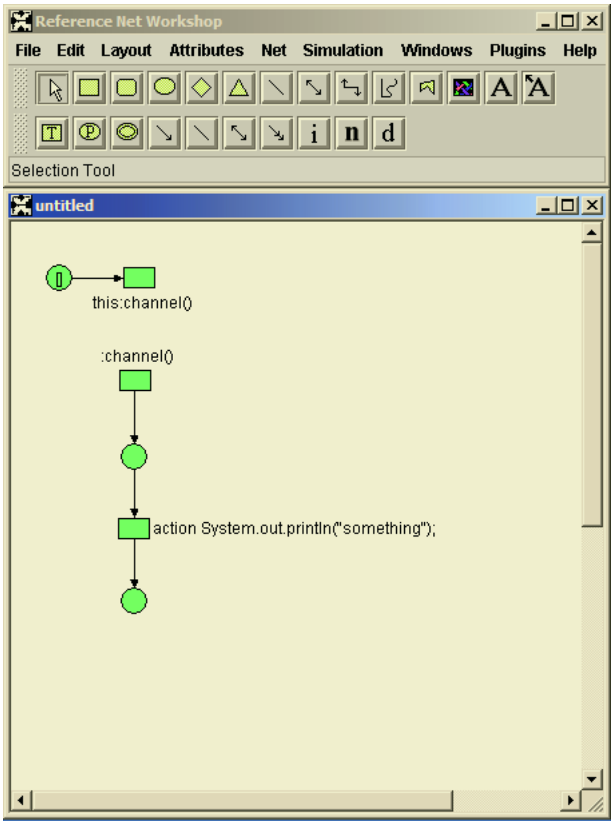

Reference Net Workshop
File Edit Layout Attributes Net Simulation Windows Plugins Help
Selection Tool
untitled
this:channel()
:channel()
action System.out.println("something");

Fig. 1. Renew

The Petri nets of a process description are assembled through templates. These templates specify the basic net layouts of the control constructs allowed in DAML-S and they contain the templates for atomic and for composite processes. These templates are placed in the order given by the process description.

To load a DAML-S description one can simply pick the file the service description is located in. If there is only one service description in this file, this description will be opened. If there are several such descriptions, the user can pick the one he or she wants to open.

4 Current Use

Within our Petri net based FIPA[4] compliant multi-agent framework Capa (see [6]), this tool can as well be used to have agents invoke Web Services dynamically. This way we have enabled a FIPA compliant multi-agent platform to interact with Web Services. Through an integration of Capa into the Agentcities.net framework we plan to offer these services to a broader community.

[4] http://www.fipa.org

Apart from them we are currently investigating in how far Web Services can be modelled through workflow techniques. Through this tool, workflows using Web Services can be executed without any need of other tools.

5 Conclusion and Outlook

We use a special kind of high-level Petri nets – reference nets – as the basic description technique and formalism for processes and workflows. The technical implementation is directly possible with the Renew-tool. Our integration into the agent domain as an open framework results by adding our FIPA-compliant extension Capa.

As the major contribution here we see the visualisation of DAML-S process descriptions as well as their execution and their integration into a FIPA compliant multi-agent system. Furthermore we see reference nets as a major modelling technique for processes. They have a formal semantics, their modelling is supported by a tool – Renew – and they have successfully been applied to a wide range of different fields.

What is missing now is the direction from reference nets which have been modeled directly or which have been derived from other models (e.g. from UML or AUML interaction diagrams (see [4])) to DAML-S descriptions. Apart from that it would be of great interest to verify certain properties of the nets generated or of those to be exported as DAML-S.

Another challenge would be the integration of ontology tools to support the development of process ontologies in DAML-S as well as the development of static ontologies in DAML+OIL. Here an integration of tools like Protege[5] or OilEd[6] is envisioned.

In the context of Petri nets workflow patterns need to be integrated into nets simplifying the creation of workflows. Here some approaches exist within the scope of agent protocols (see [4]).

References

1. Wil M.P. van der Aalst and Arthur H.M. ter Hofstede. Workflow pattern homepage. Technical report, URL: `http://tmitwww.tm.tue.nl/research/patterns/`, 2003.
2. Wil van der Aalst. Verification of workflow nets. Number 1248 in Lecture Notes in Computer Science, pages 407–426, Berlin Heidelberg New York, 1997. Springer Verlag.
3. Wil van der Aalst, Daniel Moldt, Rüdiger Valk, and Frank Wienberg. Enacting Interorganizational Workflows Using Nets in Nets. In Jörg Becker, Michael zur Mühlen, and Michael Rosemann, editors, *Proceedings of the 1999 Workflow Management Conference Workflow-based Applications, Münster, Nov. 9th 1999*, Working Paper Series of the Department of Information Systems, pages 117–136, University of Münster, Department of Information Systems, Steinfurter Str. 109, 48149 Münster, 1999. Working Paper No. 70.

[5] http://protege.stanford.edu/index.html
[6] http://oiled.man.ac.uk/

4. Lawerence Cabac, Daniel Moldt, and Heiko Rölke. A proposal for structuring petri net-based agent interaction protocols. In Wil van der Aalst and Eike Best, editors, *Proc. of 24nd International Conference on Application and Theory of Petri Nets 2003 (ICATPN 2003), Eindhoven, NL*, Berlin Heidelberg New York, 2003. to be published in Lecture Notes in Computer Science, Springer-Verlag.
5. Søren Christensen and Niels Damgaard Hansen. Coloured Petri Nets Extended with Channels for Synchronous communication. Technical Report DAIMI PB–390, Computer Science Department, Aarhus University, DK-8000 Aarhus C, Denmark, April 1992.
6. Michael Duvigneau, Daniel Moldt, and Heiko Rölke. Concurrent architecture for a multi-agent platform. In Fausto Giunchiglia, James Odell, and Gerhard Weiß, editors, *Third International Workshop, AOSE 2002, Bologna, Italy, July 15, 2002, Revised Papers and Invited Contributions*, volume 2585 of *LNCS*, Berlin Heidelberg New York, 2003. Springer.
7. Thomas Jacob. Implementierung einer sicheren und rollenbasierten Workflowmanagement-Komponente für ein Petrinetzwerkzeug. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 2002.
8. K. Jensen. High Level Petri Nets. In A. Pagoni and G. Rozenberg, editors, *Applications and Theory of Petri Nets*, number 66 in Informatik Fachberichte, pages 166–180, Berlin, 1983. Springer-Verlag.
9. Olaf Kummer. Simulating synchronous channels and net instances. In Jörg Desel, Peter Kemper, Ekkart Kindler, and Andreas Oberweis, editors, *5. Workshop Algorithmen und Werkzeuge für Petrinetze*, Forschungsbericht Nr. 694, pages 73–78. Fachbereich Informatik, Universität Dortmund, 1998.
10. Olaf Kummer. Introduction to Petri nets and reference nets. *Sozionik Aktuell*, 1:1–9, 2001. ISSN 1617-2477.
11. Olaf Kummer. *Referenznetze.* Logos Verlag, Berlin, 2002.
12. Olaf Kummer, Annette Laue, Matthias Liedtke, Daniel Moldt, and Heiko Rölke. Höhere Petrinetze zur kompakten Modellierung und Implementierung von Verhalten. In Holger Giese and Stephan Philippi, editors, *Visuelle Verhaltensmodellierung verteilter und nebenläufiger Software-Systeme, 8. Workshop des Arbeitskreises GROOM der GI Fachgruppe 2.1.9 Objektorientierte Software-Entwicklung, 13.-14. November 2000, Universität Münster*, pages 27–32, November 2000. Techreport 24/00-I.
13. Olaf Kummer, Frank Wienberg, and Michael Duvigneau. Renew – The Reference Net Workshop. WWW-Dokument unter `http://renew.de/`, 2003.
14. Srini Narayanan and Sheila A. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the eleventh international conference on World Wide Web*, pages 77–88, Honolulu, Hawaii, 2002. ACM Press.
15. R. Valk. Petri nets as token objects - an introduction to elementary object nets. In J. Desel and M. Silva, editors, *Proc. Application and Theory of Petri Nets, Lisbon, Portugal*, number 1420 in LNCS, pages 1–25, Berlin, 1998. Springer-Verlag.
16. W. M. P. van der Aalst and K. Anyanwu. Inheritance of interorganizational workflows to enable business-to-business E-commerce. In *Proceedings of the Second International Conference on Telecommunications and Electronic Commerce (ICTEC'99)*, pages 141–157. Nashville, Tennessee, October 1999.

Integrating Meta-modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation

Roswitha Bardohl[1], Hartmut Ehrig[1], Juan de Lara[2], and Gabriele Taentzer[1]

[1] Computer Science Department
Technische Universitat Berlin
Berlin, Germany
{rosi,ehrig,gabi}@cs.tu-berlin.de
[2] Escuela Politécnica Superior
Ingeniería Informática
Universidad Autónoma de Madrid
Juan.Lara@ii.uam.es

Abstract. Visual languages (VLs) play a central role in modelling various system aspects. Besides standard languages like UML, a variety of domain-specific languages exist which are the more used the more tool support is available for them. Different kinds of generators have been developed which produce visual modelling environments based on VL specifications. To define a VL, declarative as well as constructive approaches are used. The meta modelling approach is a declarative one where classes of symbols and relations are defined and associated to each other. Constraints describe additional language properties. Defining a VL by a graph grammar, the constructive way is followed where graphs describe the abstract syntax of models and graph rules formulate the language grammar.
In this paper, we extend algebraic graph grammars by a node type inheritance concept which opens up the possibility to integrate both approaches by identifying symbol classes with node types and associations with edge types of some graph class. In this way, declarative as well as constructive elements may be used for language definition and model manipulation. Two concrete approaches, the GenGED and the AToM^3 approach, illustrate how VLs can be defined and models can be manipulated by the techniques described above.

1 Introduction

Visual languages (VLs) play a central role in modelling various system aspects. One, if not the main visual modelling language is the UML [19] which integrates a number of different diagram techniques, useful to describe structural as well as behavioural aspects of object-oriented software systems. Although the UML defines a standard in visual modelling, there are of course various further visual modelling techniques, often domain-specific and often for specific aspects. Especially for those domain-specific solutions which are not widely known, a generator for visual modelling environments is useful. After specifying the VL in mind, a supporting modelling environment consisting of visual editors, simulators, compilers and animators is generated automatically and does not have to be coded by hand. Thus, rapid prototyping is supported.

There are mainly two different lines to define a VL: the declarative way and the constructive way. UML is defined by the Meta Object Facilities (MOF) approach [19] which uses classes and associations to define symbols and relations of a VL. Within this meta modelling approach, multiplicities and OCL constraints [23] are additionally used to formulate desired language properties. While constraint-based formalisms provide a declarative approach to VL definition, grammars are more constructive, i.e. closer to the implementation. In [18] for example, textual as well as graph grammar approaches are considered for VL definition. Due to its appealing visual form, graph grammars can directly be used as high-level visual specification mechanism for VLs [4]. Defining the abstract syntax of visual forms as graphs, a graph grammar defines directly the language grammar. The induced graph language determines the corresponding VL. Visual language parsers can be immediately deduced from such a graph grammar. Furthermore, abstract syntax graphs are also the starting point for model simulation and transformation, i.e., model manipulation [5, 10, 22, 13]. Also here, it is very natural to use graph transformation to come up with a high-level and constructive specification.

In this paper, we consider the integration of meta modelling with graph transformation. As common basis we take into account the types of visual symbols and relations within a VL, i.e. the visual alphabet. While constraints describe additional requirements on this alphabet, transformation rules formulate a constructive procedure. In the MOF approach, classes of symbols can be inherited, meaning that their attribute lists and their associations are also present at all their descendants. Considering graph transformation on the other hand, an additional type graph [8] is used to ensure a certain type safety on nodes and edges. Supporting node type inheritance in addition, leads to a more dense form of a graph transformation system, since similar transformation rules can be abstracted into one. We believe this work can be very valuable for the Model Driven Arquitecture [19] (MDA), where model transformation plays a central role. In Section 2, we present algebraic graph transformation with node type inheritance facilities and show how this kind of graph transformation can be flattened to simply typed graph transformation.

The MOF and the graph transformation approach can be integrated by identifying symbol classes with node types and associations with edge types. In this way, declarative as well as constructive elements may be used for language definition, but it is still open how single parts of a VL specification are defined. In Section 3, we discuss two possible approaches, the AToM^3 and the GenGED approach, which are quite different to each other.

All new concepts are illustrated at a running example which is a variant of UML Statecharts. We focus on the abstract syntax definition of the language as well as the simulation of concrete state models. Last but not least, we compare our approaches to further ones in the literature.

2 Typed Graph Transformation with Node Type Inheritance

In this section we present our new concepts of typed graph transformation with node type inheritance. Due to the space limitations, we omitted all proofs and details. The interested reader is asked to consult [2].

2.1 Type and Instance Graphs

The basic idea for specifying node type hierarchies is to introduce a special kind of (directed) edges, hierarchy edges, into type graphs. The source node of a hierarchy edge is said to be a sub-type of the target node, which is called the former one's super-type. Nodes are marked either as *concrete* or *abstract*. In host graphs only nodes of concrete types shall occur, while graphs in rules may contain nodes of both types.

Definition 1. *(Type Graph with Inheritance) A type graph with inheritance is a triple (TG, I, A) consisting of a type graph $TG = (N, E, s, t)$ (with a set N of nodes, a set E of edges, a source and a target function $s, t : E \to N$), an inheritance graph I sharing the same set of nodes N, and a set $A \subseteq N$, called abstract nodes.*
For each node n in I the inheritance clan *is defined by $clan_I(n) = \{n' \in N \mid \exists\, path\ n' \xrightarrow{*} n \text{ in } I\}$ where path of length 0 is included, i.e. $n \in clan_I(n)$.*
The sub-graph spanned by the hierarchy edges must be acyclic.

To benefit from the well-founded theory of graph transformation [8], type graphs with inheritance can be flattened to ordinary ones.

Definition 2. *(Closure of Type Graph with Inheritance) Given (TG, I, A) with $TG = (N, E, s, t)$, the* abstract closure *of (TG, I, A) is the graph $\overline{TG} = (N, \overline{E}, \overline{s}, \overline{t})$ with*

- $\overline{E} = \{(n_1, e, n_2) \mid n_1 \in clan_I(s(e)), n_2 \in clan_I(t(e)), e \in E\}$,
- $\overline{s}((n_1, e, n_2)) = n_1$,
- $\overline{t}((n_1, e, n_2)) = n_2$, *and*
- $proj_E((n_1, e, n_2)) = e$ *for* $e \in E$.

The graph $\widehat{TG} = \overline{TG}|_{N-A}$ is called concrete closure *of (TG, I, A).*
Given a graph $G = (N, E, s, t)$ and a set $X \subseteq N$, we denote by $G|_X$ the sub-graph $(X, E_X = \{e \in E \mid s(e), t(e) \in X\}, s|_{E_X}, t|_{E_X})$.

The discrimination between the abstract and the concrete closure of a type graph is necessary, since there are instance graphs with respect to either one. The left-hand side (LHS) and right-hand side (RHS) of abstract rules are typed over the abstract closure, while ordinary host graphs and concrete rules (see section 2.2 for rules) are typed over the the concrete closure. Due to the existence of the canonical inclusion $inc_{TG} : \widehat{TG} \hookrightarrow \overline{TG}$ all graphs typed over $\widehat{TG}$ are also typed over $\overline{TG}$.

Definition 3. *(Instance Graph of Type Graph with Inheritance) An abstract instance graph $(G, type)$ of (TG, I, A) is an instance graph of $\overline{TG}$, i.e. $(G, type\colon G \to \overline{TG})$. Analogously, a concrete instance graph of (TG, I, A) is typed over $\widehat{TG}$.*

The choice of triples for the edges of a type graph's closure allows expressing a typing property with respect to the type graph with inheritance. The instance graph can be typed over the type graph with inheritance (for convenience) by a pair of functions, one assigning a node type to each node and the other one assigning an edge type to each edge. Both are defined canonically. A graph morphism is not obtained this way, but some mapping that will be introduced as *clan morphism*, uniquely characterizing the type morphism into the flattened type graph.

Definition 4. *(Clan Morphism) Given a type graph with inheritance (TG, I, A), $type' \colon G \to TG$ is a* clan-morphism, *if for all $e \in G_E$ holds*
- $type'_N \circ s_G(e) \in clan_I(s_{TG} \circ type'_E(e))$ *and*
- $type'_N \circ t_G(e) \in clan_I(t_{TG} \circ type'_E(e))$.

$type'$ is called concrete, *if $type'_N(n) \notin A$ for all $n \in G_N$.*

The notion of a *type refinement* is used in order to formalize the relationship between abstract and concrete rules as they are proposed in Section 2.2. It defines an order over possible typing morphisms for a given instance graph. A typing morphism is said to be *finer* than another one, if it assigns more concrete node types to the nodes of the instance graph.

Definition 5. *(Type Refinement)*
$(G, type' \colon G \to TG)$ is a type refinement *of $(G, type \colon G \to TG)$, if*
- $type'_N(n) \in clan(type_N(n))$ *for all $n \in G_N$ and*
- $type'_E = type_E$.

$type'$ is respectively called finer *than type, denoted $type' \leq type$.*

Applying graph transformation with node type inheritance to visual language definition, usually needs attributed nodes. Thus, we have to clarify how the concept of node type inheritance can be extended to node attributes. Assuming node type A has attributes, a descendant node type B inherits not only all adjacent edge types but also its attribute list. Of course, it should be possible to enlargen the inherited list by new attributes.

If we use attributes only as labels, i.e. they are not changed during a transformation, this kind of typed attributed graphs can be defined by ordinary typed graphs. (Potentially infinite) sets of data values are considered as nodes. They are called data nodes in contrast to object nodes denoting all other nodes of an attributed graph. Data nodes and object nodes are linked by attributes, i.e. edges with an object node as source and a data node as target. We assume that there are no edges starting at some data node. If this property is satisfied within the type graph, it also holds for the instance graphs due to the typing morphisms.

Summarising, graphs and graph transformation with node attributes which are not changed are already captured by our formalisation. If we need a more general attribution concept where computations can take place on attributes, future work is needed to extent the formal approach.

Example: Type Graph for a Statechart Variant. Fig. 1 shows a type graph with inheritance for a slightly modified sub-set of the Statecharts meta model proposed in the UML specification [19]. For space limitations, the following simplifications have been performed. Only PseudoStates of the *initial* kind (attribute *ind*) are considered, i.e., we eliminated classes *SynchState*, *StubState* and concurrent states and concentrate on *CallEvent* and *SignalEvent* classes. Events are associated to the transitions they trigger (and not to states). For simulation, objects need to receive events, so we modelled an event queue (by relationships *receives* and *next*); the last event is a special one depicting its end. Additionally, we added a relationship *current* to depict the state a particular object is in. Note how the triple (TG, I, A) has been expressed in a single graph, where the nodes of TG and I are the same, regular edges represent edges in TG, hollow arrow-head edges represent edges in I and the elements of A are represented in italics.

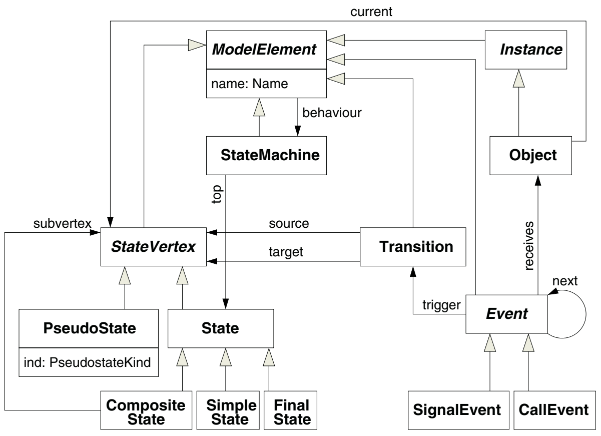


Fig. 1. Type graph with inheritance for a part of UML Statecharts.

2.2 Rules and Derivations

Transformations of graphs are described by graph rules. We follow the so-called *Double Pushout* approach to graph transformation [8]. It is desired to allow abstract node instances in rules, such that abstract rules actually represent a set of structurally similar rules, we call *concrete rules*. To get all concrete rules for an abstract rule, any combination of node replacements in the rule's LHS (being of concrete or abstract type) by instances of respective concrete sub-types (reflexive and transitive, i.e. the type's clan) must be considered. The rule morphism's image of an LHS node must always be replaced by an instance of the same type. The other nodes in the RHS remain the same and therefore must be instances of concrete types. Concrete rules are structurally equal to the abstract rule, their typing morphisms are finer (cf. Def. 5) than the ones of the abstract rule and are concrete clan morphisms.

Definition 6. *(Abstract and Concrete Rules)*
An abstract rule typed over a type graph TG with inheritance is given by $r = (L \xleftarrow{l} K \xrightarrow{r} R,\ type,\ NAC)$, where l and r are graph morphisms, type is a triple of typing clan morphisms $type = (type_L\colon L \to TG,\ type_K\colon K \to TG,\ type_R\colon R \to TG)$, and NAC is a set of triples $nac = (N, n, type_N)$ with N being a graph, $n\colon L \to N$ an injective graph morphism, and $type_N\colon N \to TG$ a typing clan morphism, such that the following conditions hold:

- $type_L \circ l = type_K = type_R \circ r$
- $type_{R,N}(R'_N) \cap A = \emptyset$, *where* $R'_N := R_N - r_N(K_N)$
- $type_N \circ n \leq type_L$ *for all* $(N, n, type_N) \in NAC$

A concrete rule r_t with respect to an abstract rule r is given by $r_t = (L \xleftarrow{l} K \xrightarrow{r} R, t, NAC)$, where t is a triple of concrete typing clan morphisms $t = (t_L\colon L \to TG, t_K\colon K \to TG, t_R\colon R \to TG)$ such that the following conditions hold (cf. Fig. 2):

- $t_L \circ l = t_K = t_R \circ r$
- $t_L \leq type_L$, $t_K \leq type_K$, $t_R \leq type_R$, *and*
- $t_{R,N}(x) = type_{R,N}(x) \forall x \in R'_N$.

The set of all concrete rules r_t with respect to an abstract rule r is denoted by $\widehat{r}$.

$$\begin{tikzcd}
N \arrow[drrr, bend right=15, "type_N"] \arrow[drrr, bend right=20, "t_{N_i}"'] & & L \arrow[ll, "n"'] \arrow[dr, "type_L"] \arrow[dr, shift right, "t_L"'] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, shift left, "type_K"] \arrow[d, shift right, "t_K"'] & R \arrow[dl, "type_R"] \arrow[dl, shift right, "t_R"'] \\
 & & & TG &
\end{tikzcd}$$

Fig. 2. Abstract and concrete rules.

The main idea for the application of an abstract rule is to apply one of its concrete rules. Both the host graph and the concrete rule are typed by concrete clan morphisms such that we can define the application of concrete rules. Later we will also define the application of an abstract rule and the equivalence of both (cf. Theorem 1).

Definition 7. *(Matching and Application of Concrete Rules)*
Let $r_t = (L \xleftarrow{l} K \xrightarrow{r} R, t, NAC)$ *be a concrete rule,* $(G, type_G)$ *a typed graph, with* $type_G\colon G \to TG$ *being a concrete clan morphism, and* $m\colon L \to G$ *a graph morphism.* m *is a* match *with respect to* r_t *and* $(G, type_G)$*, if*

- m *is a match with respect to the untyped rule* $L \xleftarrow{l} K \xrightarrow{r} R$ *and the graph* G*,*
- $type_G \circ m = t_L$*, and*
- m *satisfies the negative application conditions* NAC*, i.e. for each* $(N, n, type_N) \in NAC$ *it holds, that there does not exist a morphism* $o\colon N \to G$*, such that* $o \circ n = m$ *and* $type_G \circ o \leq type_N$*.*

Given a match m*, the concrete rule can be applied to the typed graph* $(G, type_G)$ *via* m*. A direct derivation step is denoted by* $(G, type_G) \overset{r_t,m}{\Longrightarrow} (H, type_H)$ *and can be constructed similar to the classical theory of graph transformations [8].*

In [2] we have shown that it is equivalent to apply concrete rules where typing is given by concrete clan morphisms or to apply classical rules with typing morphisms over a given type graph which is the concrete closure over a type graph with inheritance. Nevertheless, it makes sense to examine whether it is possible to find a more direct way to apply an abstract rule, because it is impractical for a tool implementing graph transformation with node type inheritance to hold all concrete rules of an abstract rule in memory or for each of them to find a match morphism into a host graph. Since abstract and concrete rules differ only in typing, but have the same structure, a match morphism from the LHS of the concrete rule into a given instance graph is also a match morphism for the abstract rule, for the latter one not being compatible with typing, though. Using the notion of type refinement, however, we can express a compatibility property.

Definition 8. *(Matching and Application of Abstract Rules)*
Let $r = (L \xleftarrow{l} K \xrightarrow{r} R, type, NAC)$ *be an abstract rule typed over* TG*,* $(G, type_G)$ *a typed graph with* $type_G\colon G \to TG$ *being a concrete clan morphism, and* $m\colon L \to G$ *a graph morphism. Then* m *is a match with respect to* r *and* $(G, type_G)$*, if*

- *m is a match with respect to the untyped rule $L \xleftarrow{l} K \xrightarrow{r} R$ and the graph G.*
- *$type_G \circ m \leq type_L$.*
- *$t_{K,N}(x_1) = t_{K,N}(x_2)$ for $t_K = type_G \circ m \circ l$ and all $x_1, x_2 \in K_N$ with $r_N(x_1) = r_N(x_2)$.*
- *m satisfies NAC, i.e. for each $nac = (N, n, type_N) \in NAC$ it holds that it does not exists a morphism $o\colon N \to G$ such that $o \circ n = m$ and $type_G \circ o \leq type_N$.*

Given a match m, the abstract rule can be applied to $(G, type_G)$ yielding an abstract direct derivation $(G, type_G) \overset{r,m}{\Longrightarrow} (H, type_H)$ with concrete type graph $(H, type_H)$ as follows:

1. *Construct the untyped direct derivation $G \overset{r,m}{\Longrightarrow} H$ in the sense of [9].*
2. *Construct $type_D$ and $type_H$ as follows*
   - *$type_D = type_G \circ l'$*
   - *$type_H(x) = \underline{if}\ x = r'(x')\ \underline{then}\ type_D(x')\ \underline{else}\ type_R(x'')$, where $m'(x'') = x$ and $x \in H_E$ or $x \in H_N$*

Theorem 1. *(Equivalence of Abstract and Concrete Direct Derivations)*
Given an abstract rule $r = (L \longleftarrow K \longrightarrow R, type, NAC)$ over a type graph TG with inheritance, a concrete typed graph $(G, type_G)$ and a structural match morphism $m\colon L \to G$ (i.e. a match with respect to the untyped rule $L \longleftarrow K \longrightarrow R$). Then the following statements are equivalent, where $(H, type_H)$ is the same concrete typed graph in both cases:

1. *$m\colon L \to G$ is a match with respect to the abstract rule r yielding an abstract direct derivation: $(G, type_G) \overset{r,m}{\Longrightarrow} (H, type_H)$.*
2. *$m\colon L \to G$ is a match with respect to the concrete rule $r_t = L \longleftarrow K \longrightarrow R$ with $r_t \in \widehat{r}$ and $t_L = type_G \circ m$ yielding a concrete direct derivation: $(G, type_G) \overset{r_t,m}{\Longrightarrow} (H, type_H)$.*

Theorem 1 allows us to use the dense form of abstract rules for model manipulation instead of generating and holding all concrete rules, i.e., abstract derivations are much more efficient than concrete derivations. In this sense, Theorem 1 allows us to use on the one hand an efficient procedure and on the other hand we are sure that the result is the same as in the classical theory using concrete rules. Moreover, as a consequence of Theorem 1, graph languages built over abstract rules and mechanisms are equivalent to graph languages that are built over a corresponding set of concrete rules. In general, rules together with a start graph define a graph grammar building up a graph language.

In the case of attribute labels, it might be convenient to add variable nodes of data types to rule graphs which are matched by concrete labels when applying such a rule. Please note that in the following figures for our example, the same variable might occur several times in a rule. It corresponds to one variable node which has to be matched by one data node. (Compare e.g. rule 2 in Fig. 3.)

Example: Generation of Statecharts. The graph grammar for generating valid Statechart instances according to the type graph with inheritance presented in Fig. 1 is shown in Fig. 3, where especially the type *StateVertex* (SV) is abstract. Please note that we omit the gluing graph K for illustrational reasons. The start graph contains a node of type

StateMachine (SM) connected to an *object* (OB). The UML specification establishes that a *StateMachine* has a unique *top* state of type *State*, but the UML well-formedness rules establish that its type should be further refined into a *CompositeState* (CS). For this purpose, rule 1 checks whether the *StateMachine SM* has already a *top* state and if this is not the case, it creates a *top* state together with a *CompositeState* (CS) and a *PseudoState* (PS) of the *initial* kind.

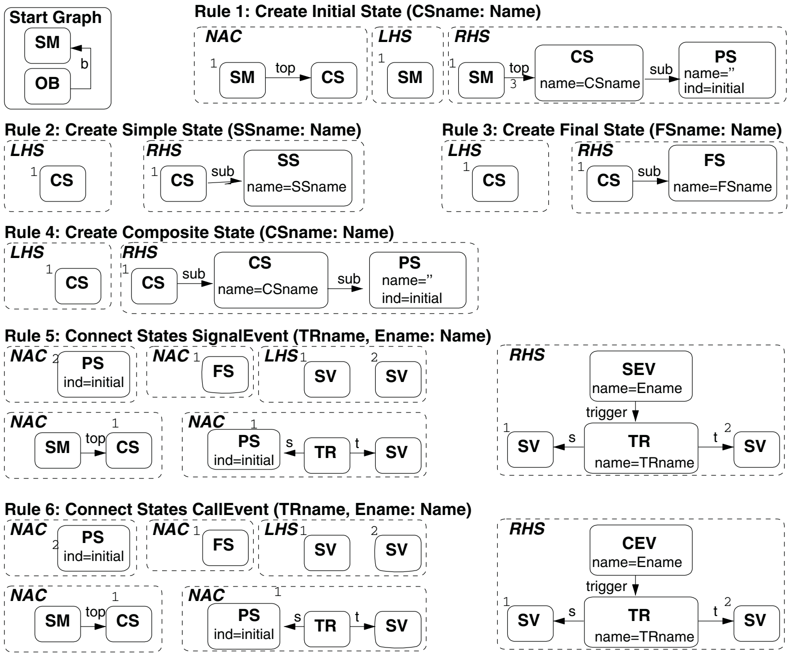


Fig. 3. Graph grammar for generating valid Statecharts.

Rules 2, 3 and 4 create new *SimpleState* (SS), *FinalState* (FS) and *CompositeState* (CS) objects inside a given *CompositeState*. In contrast to rule 1 (where the multiplicity of relationship *top* in the side of the *State* class is “1..1”) the multiplicity of the *subvertex* relationship (from *CompositeState* to *StateVertex* in the side of the latter class) is “0..*”. This implies that there is no need for a negative application condition checking the multiplicity. Additionally, each *StateVertex* should be connected to at most one *CompositeState* through relationship *subvertex*. This is achieved by the graph grammar as each newly created state is attached to a single *CompositeState*, and this relationship cannot be modified later.

Finally, rules 5 and 6 allow connecting two objects of type *StateVertex* (SV). Rule 5 describes the insertion of a transition with *SignalEvent* (SEV), while rule 6 handles the case of *CallEvent* (CEV). They are *abstract* rules as *StateVertex* is an abstract class. Additionally, the UML specification establishes (by means of constraints expressed in OCL) that a *FinalState* should not have any outgoing connection, that an *PseudoState* of the *initial* kind should not have any incoming connection and at most one outgoing connection, and that the *top* state should not have any outgoing connection. We graphically modelled these constraints by means of negative application conditions (NACs). The advantages of using abstract rules here are clear, as otherwise we would have to model rules for the valid combinations of the states we want to connect. Additionally, the typing in NACs is more concrete than the corresponding typing in the LHS.

Fig. 4 shows a Statechart obtained through the derivation of the previous graph grammar. The concrete syntax of the final Statechart is shown in the lower right corner. In the third step in the derivation, abstract rule 5 is applied. Abstract types of nodes 1 and 2 in the rule instantiate to *PseudoState PS* and *SimpleState* (node called *'SS1'* in the graph), respectively. In the example, abstract rules 5 and 6 have been applied with other instantiations to connect nodes *'SS1'* (type *SS*) and *'CS2'* (type *CS*), *'CS2'* (type *CS*) and *'FS1'* (type *FS*), *'SS2'* (type *SS*) and *'SS1'* (type *SS*), as well as *PseudoState PS* and *'SS2'* (type *SS*). Without the possibility to model abstract rules, we would have had to create concrete rules for these combinations.

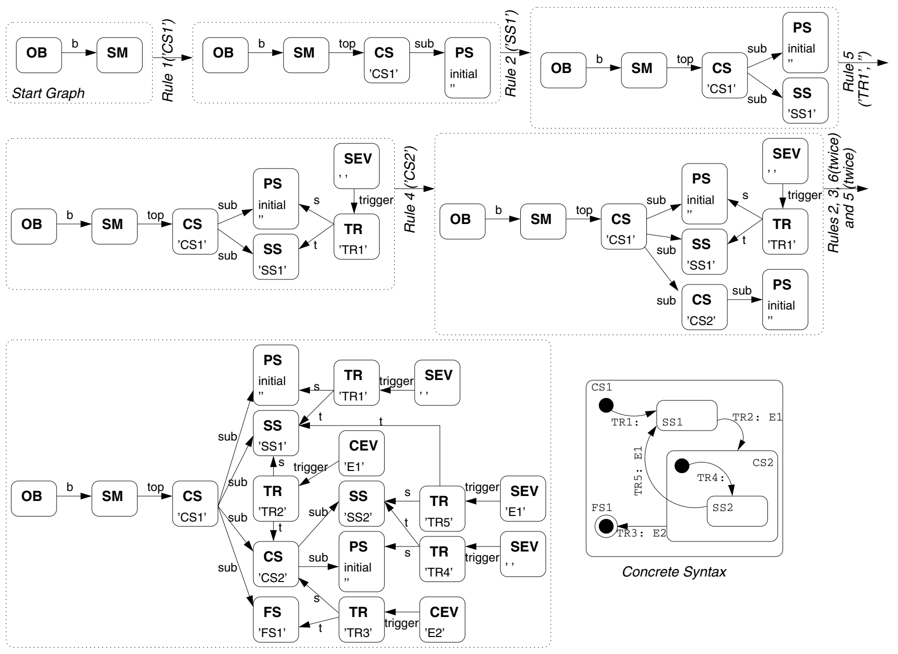


Fig. 4. A derivation of the graph grammar for generating Statecharts.

Example: Simulation of Statecharts. Fig. 5 shows a rule set for simulating our subset of Statecharts. The first rule adds the *current* relationship (*c*) to an object (OB) if it does not already have one. The initial state is the only *InitialState* node which is a *subvertex* (sub) of the *top* state. Rule 2 models a state change due to a transition from the current state. This is an abstract rule, as *StateVertex* nodes are abstract. This feature allows us to condense in a single abstract rule the combinations of all concrete subtypes of *StateVertex* nodes. Rule 3 is similar to the previous one, but models a state change into a composite state. In this case, the current state should be its initial state (that is, the *PseudoState* node is *subvertex* of the *CompositeState*). Rule 4 moves from the initial state to another one without considering events (one does not have to wait for an event to move from this *PseudoState*.) Finally, rule 5 models the fact that we can change the state due to transitions departing from any of the super-states of the *current* state. Thus, this rule allows going up in the *subvertex* hierarchy starting from the *current* state. We cannot apply this rule, if the *current* state is already a *subvertex* of the *top* state, or if the *current* state is indeed a *PseudoState* of the *initial* kind. The latter restriction is modelled by assigning type *State* (*ST*) to the *current* state in rule 5 (PseudoStates are not sub-classes of *State* but of *StateVertex*). The reason for forbidding this is that a transition in a PseudoState is still not finished, we have to end up in a node sub-class of *State*.

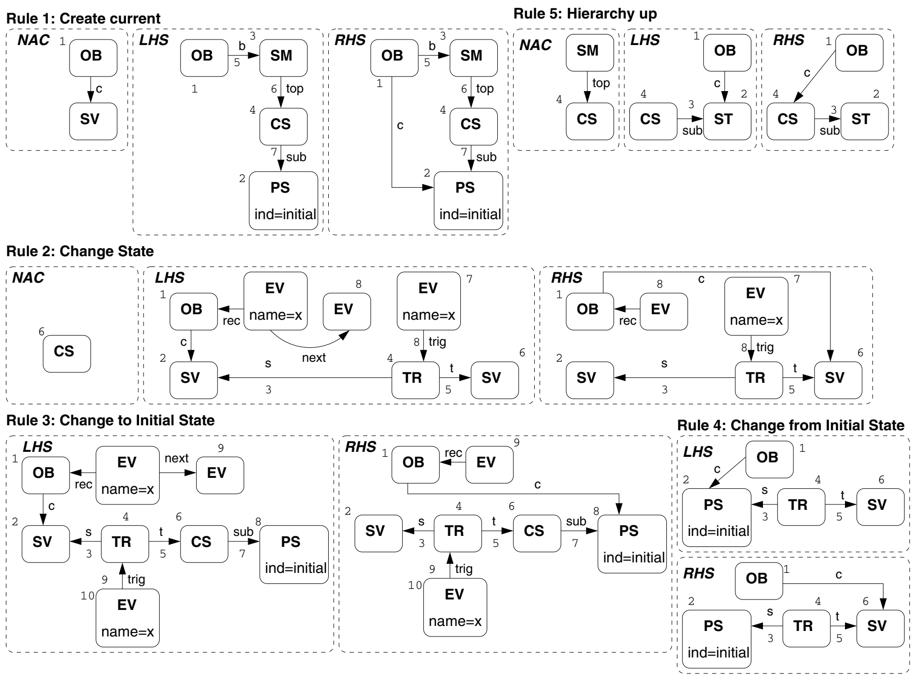


Fig. 5. Graph grammar for simulating Statecharts.

Fig. 6 shows an execution of the previous grammar to the Statechart we built in Fig. 4. In the first step, we apply rule 1, setting the *current* state pointer to the *PseudoState* (*initial* kind) of the *top* state. Then, abstract rule 4 moves the *current* state to node *'SS1'*. Node 6 in the rule (*StateVertex* type) is matched to node *'SS1'* in the graph, typed over *SimpleState*. Next, abstract rule 3 is applied and the pointer is moved to the initial state of composite state *'CS2'*. Node 2 (of type *StateVertex*) in the rule matches node *'SS1'* of type *SimpleState* in the graph; and the *Event* is of type *CallEvent*. Then, abstract rule 4 can be applied, which moves the pointer to node *'SS2'*. The type instantiation is from *StateVertex* in the rule to *SimpleState* in the graph. Now, abstract rule 5 is applied, moving the *current* pointer up in the hierarchy to node *'CS2'*. The type of node 2 (*CompositeState*) in the rule is instantiated to *SimpleState* of node *'SS2'* in the graph. For the following step, abstract rule 2 can be applied, and the pointer is set to node *'FS1'*. The type instantiation is from *StateVertex* and *Event* in the rules to *CompositeState*, *FinalState* and *CallEvent* in the graph. Here, no rule can be applied anymore, and the simulation finishes. Thus, this graph grammar models all possible simulations of the initial model. Some derivations may lead to dead ends. This may happen for example, going up in the hierarchy with rule 5, and finally discovering that none of the super-states have any outgoing transition.

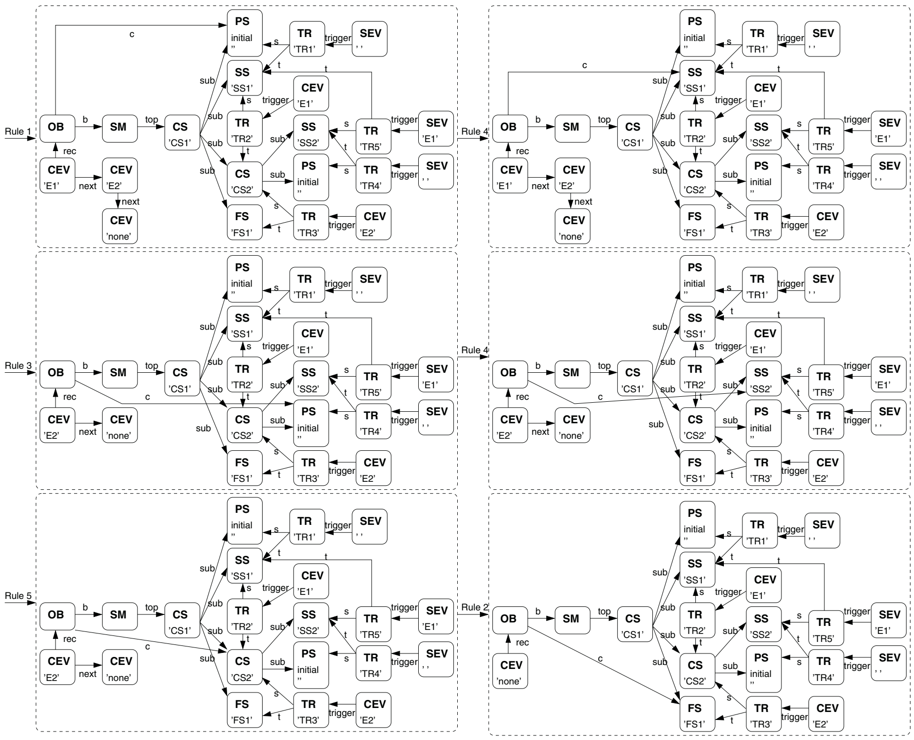


Fig. 6. A derivation of the simulation graph grammar starting from the graph generated in Fig. 4.

3 Integration of Meta-modelling with Graph Transformation

The extension of algebraic graph transformation with node type inheritance facilitates its integration with meta modelling. If we identify model element classes with node types and associations with edge types, a unique basis for the description of symbols and their relations is laid. Model elements can share common attributes and relations to other model elements which is expressed by a generalisation relationship. Similarly, an inheritance relation is supported for node types (see Sec.2). Summarising, the information expressed by class diagrams in the meta modelling approach is formulated by type graphs (with node type inheritance) for graph transformation. On top of this common basis, constraints are used to describe language properties in the meta modelling approach. On the other hand, typed graph grammars describe the modelling language as shown for the sample sub-language of Statecharts in Sec.2. In the following, two approaches for visual language (VL) definition and model manipulation are presented which distinguish in exactly this design decision. We first shortly present these approaches and compare them afterwards.

The GenGED Approach. In GenGED [1], a VL is defined (or generated) by an alphabet and a grammar. An alphabet establishes a type system for model elements (called *symbols*) and their relations (called *links*), i.e. it defines the vocabulary of a VL. The abstract syntax of symbols is represented by graph nodes, whereas graph edges represent the abstract syntax of links. The layout of symbols is given by graphical objects defining node attributes, and for each edge (abstract link) at least one graphical constraint is defined. An alphabet instance is given by an abstract syntax graph which is extended by graphical objects for the layout; the corresponding graphical constraints are solved accordingly. Usually, an abstract syntax graph is built up by VL rules (occurring in a VL grammar) which are modeled as graph rules. The grammar definition as well as the manipulation of models like Statecharts [3] is done purely by graph transformation as GenGED uses the graph transformation engine Agg [11] for this purpose.

Up to now, neither meta modelling nor inheritance concepts are realized. For defining all the features of Statecharts as we did by the type graph in Fig. 1, this type graph must be flattened in order to establish an alphabet. With the flatting, some more links have to be added. Moreover, the set of VL rules would correspond to concrete rules, i.e. the grammar contains many similar rules. Using node type inheritance concepts as proposed in Sec. 2 would reduce the set of rules in a sense that the proposed abstract rules have to be defined only. Such concise rule sets can be used to define concise abstract grammar rules for different purposes then, like syntax-directed editing, parsing, and simulation as it is supported by GenGED.

The AToM^3 Approach. AToM^3 [10] is a *multi-paradigm* modelling tool, which includes meta modelling, multi-formalism and modelling at different abstraction levels. Its main component is the *kernel*, responsible for loading, saving, creating and manipulating models, as well as for generating code for the meta modelled formalisms. The generated code must be loaded on top of the *kernel* again to allow the user building models in the defined formalism. The tool uses a pure meta modelling approach for VL definition, i.e. a VL is completely defined by a meta model, which is a type graph with inheritance with additional constraints. Some of them are assigned (pre- or post-conditions) to events (editing, connecting, etc.), the evaluation of which prohibits or enables the execution of the events and guarantees model correctness by construction.

In AToM^3, models can be manipulated by means of Python or with graph grammars. Typical manipulations are simulation, optimization and formalism transformation (which produces an instance model of a different meta model). When defining graph grammar rules, one may choose either an *"exact type matching"* or a *"sub-type matching"*. In the latter case, rules are considered *abstract* and any node can be matched with any of its sub-types. There is no distinction between *abstract* and *concrete* nodes and *sub-typing* relationships are found at runtime (by comparing nodes attributes and connections). This is due to the fact that some of the formalisms for meta modelling do not provide for inheritance. This feature also allows applying transformation rules to instances of meta models that are not explicitly related through inheritance relationships. In this way, the inheritance concept can be mapped to the semantics defined in this work, as AToM^3 can be configured to work in the *Double Pushout* approach.

Comparison of Both Approaches. After having defined the classes or types of model elements and their relations, AToM^3 supports the meta modelling approach which yields in a free editor where the model is checked according to given language constraints at specific events. Instead, GenGED can generate two kinds of editors: Either editing is done in a syntax-directed way where graph rules define the editor operations or free editing is supported where a parser has to check, if the edited model is syntactically correct. While the definition of a language by corresponding language constraints is usually easier, a parser is normally more efficient than a constraint checker. Syntax-directed editing assumes a language understanding which knows well about the structure and dependencies of its elements. Pure syntax-directed editors can be directly deduced from a language grammar. Combining both kinds of editing, the corresponding specification can be purely rule-based or mixed in the sense that rules define complex editing operations while language constraints define syntactic correctness.

Both approaches use graph transformation for model manipulations such as simulation. Due to the availability of node type inheritance, graph transformation concepts can build up directly on meta modelling concepts as in AToM^3. In GenGED, several kinds of graph transformation systems are used for different purposes as editing, parsing and simulation. Node type inheritance can condense each of them.

4 Related Work

Considering the node type inheritance concept for graph transformation, there are already tools like [21, 20] which support this concept in the same or nearly the same way. However, node type inheritance has been rarely considered in formal graph transformation approaches. The graph transformation-based language PROGRES is formalised by programmed structure rewriting systems [21] where so-called schema consistent structures are transformed. A schema corresponds to a type graph with node type inheritance, while a schema consistent structure corresponds to a well-typed instance graph. Thus, a formalisation of node type inheritance is available for PROGRES, but there is no theory building up on that. GME [16] e.g., is a meta modelling tool (for model integrated computing) which has lately incorporated graph transformation techniques for model manipulation, although its approach is not founded on the theory of graph transformation and its formalization has not been shown.

At the "Symposium on Visual Languages and Formal Methods" in 2001 there was a so-called "statechart modeling contest" where declarative as well as constructive methods have been used to define Statecharts and their behaviour. No winner was selected, but the specific strengths of the different methods have been discussed. There was not any approach integrating meta modelling with graph transformation, thus combining declarative with constructive methods. A number of graph transformation-based approaches were presented where most of the approaches could have been simplified using the hierarchy concept proposed in the present work. In addition, there is the work in [22] where Statecharts modelling is based on a meta model for extended hierarchical automata and graph transformation rules for its simulation. A similar approach is taken into account in [10] where a graph grammar is used to transform Statecharts to Petri nets which can be simulated, but there is no connection to formal graph transformation approaches.

The approach of [15] uses transformation units for generating and simulating statecharts, and is a clear example where our approach could have simplified the graph grammars. They encode the type hierarchy in graph grammar rules in such a way that they define rules for replacing each super-type for each one of its sub-types. Nonetheless, embedding conditions are needed for these rules and are not directly applicable in the standard *Double Pushout* approach.

5 Conclusions

In the literature, the main approaches to visual language definition are meta modelling and grammar-based approaches. We discussed how to integrate meta modelling with graph grammar concepts in order to support an efficient language definition and model manipulation. We presented two concrete approaches which differ in the way how meta modelling and graph transformation concepts are used and compared them.

The integration of meta modelling with graph transformation is based on a node type inheritance concept for algebraic graph transformation. This concept allows the definition of abstract rules, in which abstract nodes can appear. These can be matched with nodes of any of its sub-types. The concept is extremely useful in practice as graph grammars can be notably simplified. This has been demonstrated by showing a generation and a simulation grammar for a sub-set of UML Statecharts. The formalism presented is restricted to attributes being labels. It is up to future work to extend this work to attributed graph transformation where computations on attributes can take place and also edges may be attributed.

Moreover, analysis techniques available for attributed graph transformation such as constraint checking [14, 17] and critical pair analysis [13], should be lifted to graph transformation with node type inheritance. Having e.g. constraint checking available, language requirements could be expressed by syntactic consistency constraints in the meta modelling approach first. If parsing rules are developed thereafter, their correctness with respect to requirements could be checked. In this way we ensure that the language defined by the parser is at least a sub-language of that defined by constraints. Critical pair analysis can be useful to optimise the visual language parser (see [7]).

References

1. Bardohl, R., 2002 *A Visual Environment for Visual Languages* Science of Computer Programming 44, pages 181-203. The GenGED home page: http://tfs.cs.tu-berlin.de/genged
2. Bardohl,R., Ehrig, H., de Lara, J., Runge, O., Taentzer, G., Weinhold, I. 2003. *Node Type Inheritance Concept for Typed Graph Transformation* Technical Report 2003–19, TU Berlin.
3. Bardohl, R., and Ermel, C. 2001. *Visual Specification and Parsing of a Statechart Variant using* GenGED. In Statechart Modeling Contest, part of VLFM 2001.

4. Bardohl, R., Taentzer, G., Minas, M., Schürr, A. 1999. *Application of Graph Transformation to Visual Languages*. In Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2, H.Ehrig, G.Engels, H.-J.Kreowski, and G.Rozenberg (eds.), pages 105–181. World Scientific.
5. Baresi, L., Pezze, M. 2002. *A Toolbox for Automating Visual Software Engineering*. In FASE 2002, R. Kutsche and H. Weber (eds.), pages 189 – 202. Springer LNCS 2306.
6. Bottoni, P., Koch, M., Parisi-Presicce, F., Taentzer, G. 2001. *A Visualization of OCL using Collaborations*. In UML 2001, M.Gogolla and C.Kobryn (eds.), Springer LNCS 2185.
7. Bottoni, P., Schürr, A., Taentzer, G. 2000. *Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation*. Technical Report no. si-2000-06, University of Rome.
8. Corradini, A., Montanari, H., Rossi, F. 1996. *Graph Processes*. Special Issue of Fundamenta Informaticae, Vol 26(3-4), pages 241–266.
9. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M. 1997 *Algebraic Approaches to Graph Transformation I: Basic Concepts and Double Pushout Approach*. In Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1, G.Rozenberg (ed.), pages 163–245. World Scientific.
10. de Lara, J., Vangheluwe, H., Alfonseca, M. 2003. *Meta-Modelling and Graph Grammars for Multi-Paradigm Modelling in AToM^3*. To appear in Software and Systems Modelling. Springer. See also the AToM^3 home page at: http://atom3.cs.mcgill.ca
11. Ermel, C., Rudolf, M., Taentzer, G. 1999 *The AGG Approach: Language and Tool Environment*. In Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2, H.Ehrig, G.Engels, H.-J.Kreowski, and G.Rozenberg (eds.), pages 551 – 603. World Scientific. See also the AGG Home Page: http://tfs.cs.tu-berlin.de/agg
12. Harel, D. 1987. *Statecharts: A Visual Formalism for Complex Systems*. Science of Computer Programming, 8:231-274.
13. Heckel, R., Küster, J., Taentzer, G. 2002. *Towards Automatic Translation of UML Models into Semantic Domains*. In Proc. AGT 2002, H.-J. Kreowski (ed.), pages 11 – 22.
14. Heckel, R., Wagner, A., 1995. *Ensuring Consistency of Conditional Graph Grammars – A constructive Approach*. In ENTCS no. 2, Elsevier.
15. Kuske, S., 2001. *A Formal Semantics of UML State Machines Based on Structured Graph Transformation*. In UML 2001, M.Gogolla and C.Kobryn (eds.), Springer LNCS 2185.
16. Lédczi, A., Bakay, A., Marói, M., Vögyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G. 2001. *Composing Domain-Specific Design Environments*. IEEE Computer, pages 44-51. See also the GME home page: http://www.isis.vanderbilt.edu/Projects/gme/default.html
17. Matz, M., 2002. *Konzeption und Implementierung eines Konsistenznachweisverfahrens für attributierte Graphtransformation*. Master's thesis, TU Berlin, Fak. IV.
18. Marriot, K., Meyer, B. 1998. *Visual Language Theory*. Springer.
19. MDA, MOF and UML specifications at the OMG web page: http://www.omg.org/
20. Nickel, U., Niere, J., Zündorf, A. 2000. *The Fujaba Environment*. In ICSE 2000, pages 742–745. See also the Fujaba Home Page: http://www.fujaba.de/
21. Schürr, A. 1996. *Programmed Graph Replacement Systems*. In Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1, G.Rozenberg (ed.), pages 479–546. World Scientific. See also the PROGRES home page: http://www-i3.informatik.rwth-aachen.de/research/projects/progres/
22. Varro, D. 2002. *A Formal Semantics of UML Statecharts by Model Transition Systems*. In ICGT 2002, A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg (eds.), pages 378–392, Springer LNCS 2505.
23. Warmer, J. B., Kleppe, A. 1998. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley Object Technology Services.

An Operational Semantics for Stateflow[⋆]

Grégoire Hamon and John Rushby

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025 - USA
{hamon,rushby}@csl.sri.com

Abstract. We present a formal operational semantics for Stateflow, the graphical Statecharts-like language of the Matlab/Simulink tool suite that is widely used in model-based development of embedded systems. Stateflow has many tricky features but our operational treatment yields a surprisingly simple semantics for the subset that is generally recommended for industrial applications. We have validated our semantics by developing an interpreter that allows us to compare its behavior against the Matlab simulator. We have used the semantics as a foundation for developing prototype tools for formal analysis of Stateflow designs.

1 Introduction

The design process for embedded systems has changed dramatically over the last few years. Increasingly, designers use *model-based development environments*; these allow the system, including its software, the plant that it will control, and the environment in which it will operate, to be represented in graphical form at a high level of abstraction. Model-based development environments provide extensive tools for validation through simulation, and code generators that can compile an executable controller from its graphical representation. One of the most widely used environments of this kind is the Matlab suite from Mathworks which, with more than 500,000 licensees, is widespread throughout aerospace, automotive, and several other industries, and ubiquitous in engineering education.

Stateflow is a component of the Simulink graphical language used in Matlab. It allows hierarchical state-machine diagrams *à la* Statecharts to be combined with flowchart diagrams in a very flexible way. Stateflow is generally used to specify the discrete controller (i.e., the software) in the model of a hybrid system where the continuous dynamics (i.e., the behavior of the plant and environment) are specified using other capabilities of Simulink. As part of the Matlab tool suite, Stateflow inherits all its simulation and code generation capabilities.

⋆ This material is based on work supported by the National Science Foundation under Grant No. CCR-0086096 through the University of Illinois and by NASA Langley Research Center under Contract NAS1-00079.

The evolution to model-based development has been driven by the growing number of embedded systems, and their increasing complexity. Alongside these developments has been an increase in the criticality of embedded systems, with regard to both human safety (e.g., fly-by-wire control systems) and the cost of faults (e.g., systems deployed in huge quantities in automobiles and domestic appliances). This increasing criticality creates a need for improved methods of analysis and verification, and this provides an opportunity for formal methods. Formal methods can provide tools to check properties of a design and they can also apply a computational procedure, such as generation of test cases, systematically and automatically, to all parts of a design. However, notations like Stateflow were not built with formal methods in mind, and do not appear to be well suited to formalization.

1.1 Understanding Stateflow

Stateflow is a complex language (its User's Guide [1] is 896 pages long) with numerous, complicated, and often overlapping features lacking any formal definition. Its documentation [1, Chapter 4] describes the semantics in informal operational terms, supported by numerous examples, but the actual definition of the language is the "simulation semantics" given by its behavior when simulated in the Matlab environment. Proposing formal tools for Stateflow requires first giving it a formal definition.

This complexity of the language can be seen as an obstacle to formalization. On the other hand, it makes the need for tools to help programmers clearly visible, and users of the language are asking for them. For example, a Stateflow program can fail with a runtime exception for any of several reasons, and it is desirable to be able to avoid such failures, or at least be able to detect when a program may be vulnerable to them. One popular way to do this is to rely on programming guidelines [2,3] that restrict the language to a safe kernel. These guidelines have no more formal basis than the language itself and are based on experience. Precisely identifying the reasons for runtime errors would allow development of static analysis tools that could guarantee their absence.

1.2 A Framework for Formal Tools

In this work, we propose a formalization of Stateflow that can be used as a starting point for the definition of formal tools. Thus, we choose not to idealize the language but to follow strictly the simulation semantics given by the Mathworks documentation and tools, even in its shortcomings. The main result lies in understanding that although Stateflow is superficially similar to other statecharts notations, it is in truth a sequential imperative language. As such, the problems arising when formalizing the language are different in nature than those for other statechart languages, and different solutions are required. We use a formal operational semantics as it precisely captures the order of execution of the different components of a Stateflow chart. This operational approach satisfies our goal and is able to express the more complicated features of the language where alternative approaches (e.g., denotational) might get lost.

We have used this formalization in the development of several tools for Stateflow; it provides a detailed understanding of the language, and readily supports the construction of static analyzers and translation to formal tools such as model checkers.

1.3 Overview of the Paper

We first introduce Stateflow through an example. Section 3 develops a formalization of a subset of the language and gives it an operational semantics. Finally, in Section 4 we compare our approach with related work and sketch why the approach proposed here seems to be a good basis for developing formal tools for the language.

2 Introduction to Stateflow

The Stateflow language provides hierarchical state machines, similar to those of Statecharts (although these two languages give different semantics to the state machines). It includes complicated features like interlevel transitions, complex transitions through junctions (which are portrayed as small circles), and event broadcasting. Stateflow also provides flowcharts, which are specified using internal transitions leading to terminal junctions. Describing the whole language is beyond the scope of this paper, so we present here a simple example program that includes both kinds of notation and sketch its execution.

2.1 A Stopwatch in Stateflow

Figure 1 presents the Stateflow specification of a stopwatch with lap time measurement. This stopwatch contains a counter represented by three variables (`min`, `sec`, `cent`) and a display, also represented as three variables (`disp_min`, `disp_sec`, `disp_cent`).

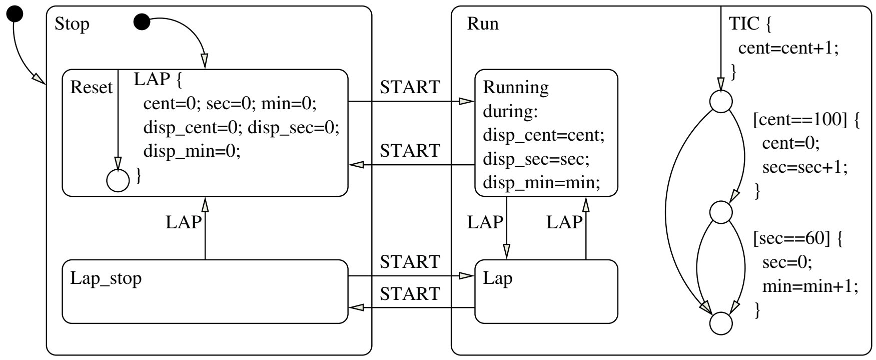


Fig. 1. A simple stopwatch in Stateflow

The stopwatch is controlled by two command buttons, START and LAP. The START button switches the time counter on and off; the LAP button fixes the display to show the lap time when the counter is running and resets the counter when the counter is stopped. This behavior can be modeled as four exclusive states:

– Reset: the counter is stopped. Receiving LAP resets the counter and the display, receiving START changes the control to the Running mode.
– Lap_Stop: the counter is stopped. Receiving LAP changes to the Reset mode and receiving START to the Lap mode.
– Running: the counter is running, and the display updated. Receiving START changes to the Stop mode, pressing LAP changes to the Lap mode.
– Lap: the counter is running, but the display is not updated, thus showing the last value it received. Receiving START changes to Lap_Stop, receiving LAP changes to Running.

These four states are here grouped by pairs inside two main states: Run and Stop, active respectively when the counter is counting or stopped. The counter itself is specified within the Run state as a flowchart, incrementing its value every time a clock TIC is received (every 1/100s).

2.2 Executing the Stopwatch Chart

A Stateflow chart always has one active state. Executing the chart consists in executing the active state each time an event occurs in the environment. Events here are either an action on one of the buttons (START or LAP) or a clock tick (TIC). Executing the active state is done in three steps:

1. See if a transition leaving the state can be taken, else goto step 2.
2. Execute internal actions (internal transitions, then during actions).
3. Execute any internal state that is active.

Transitions can be guarded by events or conditions or both, and they can trigger actions. The internal transition in state Reset for example is guarded by the LAP event and triggers a series of actions reinitializing the counter and the display. Supposing that the Run state is active, with the Running substate active, receiving the START event would trigger the following sequence of reactions:

– there is no transition leaving the state (the transitions guarded by start belong to its substates),
– the flowchart is executed, but is guarded by TIC, thus does nothing,
– the active substate is executed, it has a transition which can be fired, leading to Reset, itself substate of Stop; Running then Run are exited, and Stop then Reset are entered.

This step is completed, and execution will continue from the newly active state next time an event is received from the environment.

The model contains a flowchart that implements the counter. Flowcharts are described using transitions between junctions. Unlike states, a junction is exited instantaneously when entered, and the flowchart executes until a terminal junction (a junction without outgoing transitions) is reached, or all paths have failed. Backtracking can occur if a wrong path is tried. In our example, the flowchart is guarded by the `TIC` event. If activated under this event, the `cent` variable is incremented and the first junction reached. Two transitions leave it, the guarded one is always executed first. If `cent` is equal to 100, the guarded transition is taken, `cent` initialized to 0 and `sec` incremented, the second junction is reached, and execution continues. If `cent` is not equal to 100, the guarded transition fails, the unguarded one is tried and, being unguarded, succeeds, leading to the third junction, which is terminal, so execution ends.

This short example does not present all Stateflow features, but it introduces hierarchical states, interlevel transitions, and mixed design with flowcharts. Our informal description of the execution of this example is actually close to the presentation of the language's semantics in its documentation.

3 Formalizing Stateflow

Studying the language, we came to realize that, although superficially similar to other statechart notations, Stateflow greatly differs from them. In particular, all possibilities of non-determinism are avoided by relying on strict ordering rules, and the scheduling between concurrent components is always statically known. Thus, we decided to consider Stateflow as an imperative language, and to use a structural operational semantics (SOS) [4], which is well-adapted to the description of such languages. This semantics is efficient in dealing with the complexity of Stateflow, which lies in the intricacy of its constructions, not in concurrency or non-determinism.

3.1 A Stateflow Subset

We now introduce a linear language that is a strict subset of Stateflow. This language eliminates some difficulties of the graphical notation, by making the order between components explicit (we describe translation from the graphical form below). We then give this language a formal semantics.

The Language – The language is presented in Figure 2. Its basic components are states s, junctions j, events e, actions a, and conditions c. We also define active states s_a (nothing or a state), transition events e_t (nothing or an event), and paths (lists of states).

Transitions t are guarded by a transition event and a condition, can execute two actions and go to a destination d (either a path or a junction). The first action is executed as soon as the transition is valid; the second one is executed only if taking the transition leads somewhere.

Transitions are grouped into lists T. Junction definition lists J associate lists of transitions to junctions. State definition lists SD associate state definitions sd to states. A state definition is a triplet of actions, executed respectively upon entering, executing and exiting the state, an internal composition, a list of inner transitions, a list of outgoing transitions, and a junction definition list. Finally, a composition C is a composition of states, and is either an `And` or an `Or` composition. An `And` composition is defined by a boolean (true if the composition is active) and a state definition list. An `Or` composition is an active state, a path, a set of default transitions, and a state definition list.

composition	$C = \texttt{Or}(s_a, p, T, SD) \mid \texttt{And}(b, SD)$
state definition	$sd = ((a, a, a), C, T_i, T_o, J)$
state definition list	$SD = \{s_0 : sd_0; ...; s_n : sd_n\}$
junction definition list	$J = \{j_0 : T_0; ...; j_n : T_n\}$
transition	$t = (e_t, c, a, a, d)$
transition list	$T = \emptyset_T \mid t.T$

state	s	active state	$s_a = \emptyset_s \mid s$
junction	j		
path	$p = \emptyset_p \mid s.p$	destination	$d = p \mid j$
event	e	transition event	$e_t = \emptyset_e \mid e$
action	a	condition	c

Fig. 2. The language

Notes on the Language

- Actions a and conditions c are expressions of the *action language*, which is distinct from Stateflow itself; we keep this distinction here. The action language is a very simple imperative language. For the same reason we do not have variables here, they are part of the action language, not of Stateflow itself.
- Transition list T and state definition lists SD are ordered, and their order is significant. When using the graphical representation of a program, the order is determined by the position of the components on the chart: states are ordered top to bottom and then left to right. Transitions are ordered following the *12 o'clock rule*: they are first ordered using a partial ordering on the form of their guards (transitions guarded by an event are evaluated before transitions guarded only by a condition, and unguarded transitions come last), and when this ordering fails, they are ordered by following their source clockwise starting from a 12 o'clock position.
- In the following, state definitions will be written (A, C, nT_i, T_o, J), with A representing a triplet of actions: the `entering`, `during`, and `exit` actions will be noted respectively as $A.e$, $A.d$, and $A.x$.

An Example – The `Stop` state from the stopwatch:

$$\begin{array}{l}
\texttt{Stop}\text{: } ((\diamond,\ \diamond,\ \diamond), \\
\quad\quad \texttt{Or}(\emptyset_s, \texttt{Stop},\ (\emptyset_e,\ \diamond,\ \diamond,\ \diamond,\ \texttt{Stop.Reset}).\emptyset_T, \\
\quad\quad\quad \{\ \texttt{Reset}\text{: } ((\diamond,\ \diamond,\ \diamond),\ \texttt{Or}(\emptyset_s, \texttt{Stop.Reset},\ \emptyset_t,\ \{\ \}), \\
\quad\quad\quad\quad\quad\quad\quad\quad (\texttt{LAP},\ \diamond, \\
\quad\quad\quad\quad\quad\quad\quad\quad\ (\texttt{cent} \leftarrow 0;\ \texttt{sec} \leftarrow 0;\ \texttt{min} \leftarrow 0; \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \texttt{disp\_cent} \leftarrow 0;\ \texttt{disp\_sec} \leftarrow 0;\ \texttt{disp\_min} \leftarrow 0),\ \diamond,\ j).\emptyset_T, \\
\quad\quad\quad\quad\quad\quad\quad\quad (\texttt{START},\ \diamond,\ \diamond,\ \diamond,\ \texttt{Run.Running}).\ \emptyset_T,\ \{j : \emptyset_T\ \}); \\
\quad\quad\quad\quad \texttt{Lap\_stop}\text{: } ((\diamond,\ \diamond,\ \diamond),\ \texttt{Or}(\emptyset_s,\ \texttt{Stop.Lap\_stop},\ \emptyset_T,\ \{\ \}), \emptyset_T, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad (\texttt{LAP},\ \diamond,\ \diamond,\ \diamond,\ \texttt{Stop.Reset}). \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad (\texttt{START},\ \diamond,\ \diamond,\ \diamond,\ \text{Run.Lap}).\ \emptyset_T,\ \{\ \})\ \ \}), \\
\quad\quad \emptyset_T,\ \emptyset_T,\ \{\ \})
\end{array}$$

The $\diamond$ symbol represents both an empty action and an empty condition. The name j corresponds to the terminal junction found in state `Reset` (junctions being anonymous in Stateflow, they are given unique ids during the translation).

We see here that `Stop` is a state, containing an `Or` composition made from states `Reset` and `Lap_Stop`. `Reset` contains an internal transition guarded by `LAP` and a transition guarded by `START` going to state `Run.Running`; `Lap_Stop` contains two transitions, one guarded by `LAP` going to state `Stop.Reset`, the other guarded by `START` going to state `Run.Lap`.

3.2 Operational Semantics

Executing a Stateflow program consists, on each (discrete) step, in processing an input event through the program. This processing can modify the value of variables in the environment, raise output events, and change the program itself as it may change the active states if transitions occur.

We propose here an SOS semantics for the language. This semantics precisely expresses the sequence of actions involved in processing an event through a chart. It is made of rules with the following general form:

$$e, D \vdash P \overset{D'}{\to} P', tv$$

Processing an event e in an environment D through a program component P produces a new environment D', a new program P', and a transition value tv. P can here denote any syntactic class of the language. Transition values tv are used for communication between different parts of the chart. The rules for some of the particular syntactic classes given below extend and slightly differ from this general form.

Environments D contain bindings from variables of the action language to values and the list of output events that are raised in the current instant.

Definition 1 (Environment D).

$$D ::= [x_0 : v_0; ...; x_n : v_n; e_0; ...e_k]$$

Transition values indicate if a transition has fired or not. If no transition has fired, two distinct values, `End` and `No`, are necessary to distinguish a failing transition from the final transition of a flowchart. If a transition has fired, we keep track of its destination and of an eventual pending action.

Definition 2 (Transition value *tv*).

$$tv ::= \texttt{Fire}(d, a) \mid \texttt{End} \mid \texttt{No}$$

As for the definition of the language, we do not detail the semantics of actions and conditions here but consider that we have semantics rules such that

$$e, D \vdash a \hookrightarrow D' \qquad\qquad e, D \vdash c \rightarrow b$$

Evaluating an action when processing event e in environment D produces a new environment D'; evaluating a condition when processing e in D produces a boolean value b.

We now present the semantic rules for the different syntactic classes. For brevity we only detail rules for transitions, transition lists and parallel compositions, the full rules are available in an appendix.

Transitions (Figure 3) – A transition (e_0, c, a_c, a_t, d) fires to destination d if e_0 corresponds to the processed event e or is empty, and if the condition c is true. In this case, the action a_c is immediately executed and a_t is left pending in the returned value (rule t-Fire). If e_0 is different from the processed event and is not empty (rule t-No$_1$) or if the condition is false (rule t-No$_2$), the transition fails and returns `No`.

$$t\text{-}\textsc{Fire}\quad \frac{(e = e_0) \vee (e_0 = \emptyset_e) \qquad e, D \vdash c \rightarrow \mathit{true} \qquad e, D \vdash a_c \hookrightarrow D'}{e, D \vdash (e_0, c, a_c, a_t, d) \xrightarrow{D'} \texttt{Fire}(d, a_t)}$$

$$t\text{-}\textsc{No}_1\quad \frac{(e \neq e_0) \wedge (e_0 \neq \emptyset_e)}{e, D \vdash (e_0, c, a_c, a_t, d) \xrightarrow{D} \texttt{No}} \qquad t\text{-}\textsc{No}_2\quad \frac{(e = e_0) \vee (e_0 = \emptyset_e) \qquad e, D \vdash c \rightarrow \mathit{false}}{e, D \vdash (e_0, c, a_c, a_t, d) \xrightarrow{D} \texttt{No}}$$

Fig. 3. Rules for transitions t

Transition Lists (Figure 4) – Lists of transitions, together with junctions, are used to model both flowcharts and complex transitions between states. The important point here is that a list of transitions is processed sequentially and the first transition that can fire is taken, as shown by rules T-No and T-Fire.

If a transition fires to a junction, the list of transitions associated with this junction needs to be processed: evaluation continues instantaneously when reaching a transition. This goes on until we fire to a path (rule T-Fire-j-F), we reach a terminal junction (rule T-End) or we fail, in which case we have to backtrack and try the next transition in our first list (rule T-Fire-j-N).

$$\emptyset_t\ \frac{}{e, D, J \vdash \emptyset_t \stackrel{D}{\to} \texttt{End}} \qquad \text{T-Fire}\ \frac{e, D \vdash t \stackrel{D'}{\to} \texttt{Fire}(p, act)}{e, D, J \vdash t.T \stackrel{D'}{\to} \texttt{Fire}(p, act)} \qquad \text{T-No-Last}\ \frac{e, D \vdash t \stackrel{D_1}{\to} \texttt{No}}{e, D, J \vdash t.\emptyset_T \stackrel{D_1}{\to} \texttt{No}}$$

$$\text{T-No}\ \frac{T \neq \emptyset_T \qquad e, D \vdash t \stackrel{D_1}{\to} \texttt{No} \qquad e, D_1, J \vdash T \stackrel{D_2}{\to} \texttt{tv}}{e, D, J \vdash t.T \stackrel{D_2}{\to} \texttt{tv}}$$

$$\text{T-Fire-j-F}\ \frac{e, D \vdash t \stackrel{D_1}{\to} \texttt{Fire}(j, a_1) \qquad e, D_1, J[j : T_j] \vdash T_j \stackrel{D_2}{\to} \texttt{Fire}(p, a_2)}{e, D, J[j : T_j] \vdash t.T \stackrel{D_2}{\to} \texttt{Fire}(p, a1; a2)}$$

$$\text{T-End}\ \frac{e, D \vdash t \stackrel{D_1}{\to} \texttt{Fire}(j, a_1) \qquad e, D_1, J[j : T_j] \vdash T_j \stackrel{D_2}{\to} \texttt{End}}{e, D, J[j : T_j] \vdash t.T \stackrel{D_2}{\to} \texttt{End}}$$

$$\text{T-Fire-j-N}\ \frac{e, D \vdash t \stackrel{D_1}{\to} \texttt{Fire}(j, a_1) \qquad e, D_1, J[j : T_j] \vdash T_j \stackrel{D_2}{\to} \texttt{No} \qquad e, D_2, J[j : T_j] \vdash T \stackrel{D_3}{\to} \texttt{tv}}{e, D, J[j : T_j] \vdash t.T \stackrel{D_3}{\to} \texttt{tv}}$$

Fig. 4. Rules for transition list T

State Definitions – The rules exhibit their order of execution. Different rules are necessary for entering, executing, and exiting a state. When executing a state, outgoing transitions are tested first; if they fail, the `during` code is executed, then the internal transitions and then the internal composition. If the composition fires, the transition actions are executed followed by the exit code. If the outgoing transitions fire, the transition actions are executed, the internal composition exited, and the exit code executed.

Or Compositions – Rules for Or compositions take care of the control changes between states and handle interlevel transitions. The currently active state is executed. If this state fires, either it fires to one of its siblings, in which case this sibling is entered and becomes the active state, or it fires outside of the composition.

And Compositions (Figure 5) – Executing an And composition consists in sequentially entering/executing/exiting all its parallel substates, each state being executed in the environment returned by the execution of its predecessor. It is important to notice here that the parallel construction is in fact completely sequential, and the order of execution statically known: none of the problems associated with concurrency appears here.

And

$$\frac{(tv = \texttt{No}) \vee (tv = \texttt{End}) \qquad \forall i \in [0..n],\ \ e, D_i, J \vdash sd_i \overset{D_{i+1}}{\rightarrow} sd'_i, \texttt{No}}{e, D_0, J, tv \vdash \texttt{And}(\{s_0 : sd_0; ...; s_n : sd_n\}) \overset{D_{n+1}}{\rightarrow} \texttt{And}(\{s_0 : sd'_0; ...; s_n : sd'_n\}),\ \texttt{No}}$$

And-Init

$$\frac{p_j = p \qquad \forall i \neq j, p_i = \emptyset_p \qquad \forall i \in [0..n],\ \ e, D_i, p_i \vdash sd_i \overset{D_{i+1}}{\Uparrow} sd'_i}{e, D_0, s_j.p \vdash \texttt{And}(\{s_0 : sd_0; ...; s_n : sd_n\}) \overset{D_{n+1}}{\Uparrow} \texttt{And}(\{s_0 : sd'_0; ...; s_n : sd'_n\})}$$

And-Exit

$$\frac{\forall i \in [0..n],\ \ e, D_i \vdash sd_{n-i} \overset{D_{i+1}}{\Downarrow} sd'_{n-i}}{e, D_0 \vdash \texttt{And}(\{s_0 : sd_0; ...; s_n : sd_n\}) \overset{D_{n+1}}{\Downarrow} \texttt{And}(\{s_0 : sd'_0; ...; s_n : sd'_n\})}$$

Fig. 5. Rule for AND compositions

3.3 Supporting Local Events

We now extend our treatment to include one of the trickiest features of Stateflow, the local events mechanism, which the preceding semantics does not consider. This mechanism allows actions to send an event to a state; when this occurs, the current processing is interrupted while the sent event is processed through the receiving state. The receiving state acts here as a function, the action of sending it an event being the function call. However, this mechanism also introduces some complicated cases and fully supporting it in the general case appears difficult. We exhibit a restricted form of this mechanism that is both expressive and supports a simple semantics.

We first try to extend our semantics with a simple interpretation of local events. The action $\texttt{send}(e, s)$ sends event e to a named state s (broadcasting an event to the whole chart consists in sending an event to the `main` state). Its behavior can be expressed by the following rule:

Send

$$\frac{e', D[s : sd], \emptyset_J \vdash sd \overset{D'}{\rightarrow} sd', tv}{e, D[s : sd] \vdash \texttt{send}(e', s) \hookrightarrow D'[s : sd']}$$

Sending e' to state s results in processing it through the definition of s. We have extended environments by the definitions of states:

Definition 3 (Environment D).

$$D ::= [x_0 : v_0; ...; x_n : v_n; e_0; ...e_k; SD]$$

where SD is a list of state definitions. The notation $D[s : sd]$ denotes the environment D in which s is associated to sd.

However, this rule alone does not fully handle event sending; deeper modifications of the semantics are needed. Processing the local event changes the definition of the destination state (in the rule, the definition of `s` is `sd'` after processing the event). The destination state can be an ancestor of the current state, which might have been modified. It is necessary, whenever an action is performed, to read the (eventually new) definition of the current state and continue the execution at the corresponding control point in this new definition. If the active state has been modified by the call, the return point may even not be active anymore, which leads in Stateflow to a runtime error.

Investigating this mechanism to understand its behavior and its expressive power, we distinguished two different usages:

– Describing recursive behaviors. Recursion occurs if the caller sends an event to itself or one of its ancestors. In practice, those recursions are very difficult to control (the event sending action might get executed by the recursive call) and to understand. Providing tools to check that the recursion will stop is difficult (see [5]). Moreover, these recursions easily lead to runtime errors and their use is discouraged in industrial applications.
– Explicit scheduling of parallel states. Parallel states are normally ordered statically given their position on the chart. Local events can be used to make some explicit, or dynamic, scheduling of parallel states, guarding the states by local events and having a caller that executes them in the expected order. This particular use is much simpler to understand.

Our proposition is to limit the use of local events to the definition of sequencing behaviors. This can be obtained by imposing the following restrictions:

– Local events can be sent only to parallel states.
– Transitions out of parallel states are forbidden (this is already imposed by Stateflow, see Section 3.4 for more details).
– Loops in the broadcasting of events are forbidden (i.e., if state A broadcasts an event to state B, B cannot in turn broadcast an event to A).

Given those limitations[1] sending an event can really be seen as a function call. Forbidding transitions out of parallel states ensures that context modifications are kept local to the destination. Forcing sending to parallel states and forbidding loops ensures that no infinite calls will occur. The rule for sending an event is

[1] To keep equivalence with Stateflow, we further impose that local events can be sent only to already-visited states; this is due to initialization problems in Stateflow itself.

the rule presented before. In addition to this, we need to change only the rule for parallel execution:

$$\textsc{And}\quad \frac{D_0 = D[s_0 : sd_0; ...; s_n : sd_n] \qquad \forall i \in [0..n],\ \ e, D_i, \{\} \vdash D_i(s_i) \xrightarrow{D'_i} sd'_i, \texttt{No} \quad D_{i+1} = D'_i[s_i : sd'_i]}{e, D \vdash \texttt{And}(\{s_0 : sd_0; ...; s_n : sd_n\}) \xrightarrow{D_{n+1}\backslash\{s_0,...,s_n\}} \texttt{And}(\{s_0 : D_{n+1}(s_0); ...; s_n : D_{n+1}(s_n)\}),\ \texttt{No}}$$

The rule is similar to the original one, with the addition of the state definitions in the environment, where they are updated during execution.

This definition of local events in our opinion captures the most interesting of their uses in Stateflow, supports a simple semantics, and does not introduce new runtime error or infinite loop possibilities. The Ford guideline for Stateflow [2] makes use of local events in this exact same way.

3.4 Additional and Unsupported Features

Our subset supports nearly the whole language with the restrictions on local events presented above. The only interesting feature still missing is the history junction mechanism that keeps track of the configuration a state was in before it was last exited, and re-enters it in this configuration. Our semantics easily extends to support this mechanism; we omitted it here for the sake of simplicity. The necessary modifications are to add a history component (a boolean) in the state definitions to determine whether they carry such junctions, and to add rules to handle this component when entering and leaving states and compositions.

Two restrictions are also imposed on transitions: transitions out of a parallel compositions and interlevel transitions going to a junction are forbidden. Transitions directly out of a parallel state are already forbidden in Stateflow, but can be simulated by taking an interlevel transition from a substate of a parallel state. The behavior of such transitions is quite unpredictable, and introduces possible runtime errors (e.g., two states fire simultaneously out of the composition to different destinations). Forbidding interlevel transitions to a junction allows the semantics to be local. When taking an interlevel transition to a state, pending actions can be executed and the state closed before entering the destination. If the transition goes to a junction, we cannot be sure that it is leading somewhere, and cannot close the state before opening the destination.

3.5 Equivalence with the Simulation Semantics

Our language is intended to be a strict subset of Stateflow, so that tools developed for it will apply to programs designed using Mathworks' tools – as long as the programs are within our subset (which is checked by a tool, see Section 4.2).

For this to succeed, the semantics we are using and the simulation semantics of Stateflow must be equivalent on our subset. Our semantics was conceived with this goal in mind, precisely following Stateflow documentation but, because the simulation semantics is not formal, it is not possible to prove this equivalence. However, our SOS semantics is directly executable and can easily be used to define a Stateflow interpretor whose outputs can be compared to those from the Matlab simulator. We have done this and systematically examined many examples; for all these examples, the traces obtained by the two tools were the same.

4 Conclusion and Related Work

We have presented an operational semantics for the Stateflow language. Our semantics covers virtually the whole language, excluding only those features that are generally discouraged in industrial applications [2]. A formal semantics is the necessary basis for building formal tools for the language. The operational approach chosen here leads to a surprisingly simple semantics and thus constitutes a good starting point for such developments.

4.1 Related Work

Little work has directly addressed the semantics of Stateflow. A natural idea when considering Stateflow is to evaluate work on formalization of Statecharts [6]. However, the two languages have very different semantics (and Stateflow also includes flowcharts), so denotational approaches proposed for Statecharts semantics do not easily or usefully adapt to Stateflow.

A popular approach to Statecharts semantics is to translate the language into a simpler formalism for which a semantics is already known. This approach was followed by Mikk et al. for Statemate [7] by translation to hierarchical automata. Their semantics was adapted to UML-Statecharts by Gnesi, Latella and Massink [8]. A similar semantics was proposed for Stateflow by Tiwari, Shankar and Rushby [5] by translation to push-down automata. However, encoding the complex Stateflow language constructions requires introduction of a vast number of control variables that make using the translation by formal tools difficult.

Lüttgen, von der Beeck, and Cleaveland [9] have proposed an SOS semantics for a subset of Statecharts. They wanted to define a compositional semantics and do not consider interlevel transitions. We can notice the same effect here: although our semantics is not compositional (the language contains absolute reference to states), it can be made compositional by forbidding interlevel transitions. They also need to consider execution on a micro and on a macro level, which is not necessary here due to the completely deterministic nature of Stateflow.

The appeal of the proposed SOS semantics for Stateflow, and what makes it work, is that it exhibits the sequential behavior of Stateflow: the language does not have true concurrency nor any kind of nondeterminism. Seeing Stateflow as an imperative language, the choice for an operational approach is natural, and has the advantage of scaling well to a rich language allowing a big subset to be considered.

A similar approach is implicit in the work of Banphawatthanarak, Krogh, and Butts [10], who describe a translator from Stateflow into the input language of the SMV model checker. Although they do not construct an explicit semantics, the considerations that guide their translation are very close to ours and reflect a similar focus on the sequential nature of Stateflow execution and the importance of accurately representing its sequencing rules.

4.2 A Good Basis for Formal Tools

Our goal was to propose a formalization of Stateflow that would constitute a good foundation for construction of formal tools for the language. The presented semantics appears to meet this goal very well: while sometimes large, the rules of the semantics are simple and syntax directed, which makes them well adapted to automatic processing.

One kind of tool in which we are interested is static analysis for detecting flaws in programs and to enforce or enhance programming guidelines such as [2]. The proposed semantics, by giving a low-level view of a program's execution makes it possible to understand causes of runtime errors (through missing rules in the semantics). We have developed such a tool that checks for possible runtime errors and also detects non fatal flaws, such as possible backtracking or reliance on the 12 o'clock rule. Having a syntax-directed semantics allows a precise diagnosis to be given to the user. This tool also verifies that a program lies within the subset considered by our semantics.

We are also interested in model checking, which can be used to check properties of programs and to automate test case generation [11,12]. Our operational semantics provides a basis for efficiently compiling Stateflow to an imperative language or to the input language of a model checker. We have developed a translator to the SAL language used by SRI's model checkers; the translation produces efficient code similar in size to the Stateflow model. The SAL translation can be used to check properties of a design: our example (Figure 1) contains a bug which the model-checker easily finds (updating the display is only done when staying at least one instant in the `Running` state; if several `LAP` and `START` events occur between two `TIC`s, the display can show an erroneous value). We are currently using this translation to do automatic test-case generation for Stateflow.

In future work, we plan to investigate the formalization of the whole Simulink/Stateflow environment. One possible direction is to combine this work with existing work on Simulink [5,13].

References

1. The Mathworks: Stateflow and Stateflow Coder, User's Guide. Release 13sp1 edn. (2003)
2. Ford: Structured analysis and design using Matlab/Simulink/Stateflow - modeling style guidelines. Technical report, Ford Motor Company (1999) Available at `http://vehicle.me.berkeley.edu/mobies/papers/stylev242.pdf`.

3. Buck, D., Rau, A.: On modelling guidelines: Flowchart patterns for Stateflow. Softwaretechnik-Trends **21** (2001)
4. Plotkin, G.: A structural approach to operational semantics. Technical Report DAIMI-FN-19, Aarhus University (1981)
5. Tiwari, A., Shankar, N., Rushby, J.: Invisible formal methods for embedded control systems. Proceedings of the IEEE **91** (2003) 29–39
6. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming **8** (1987) 231–274
7. Mikk, E., Lakhnech, Y., Petersohn, C., Siegel, M.: On formal semantics of Statecharts as supported by Statemate. In: 2nd BCS-FACS Northern Formal Methods Workshop, BCS-EWIC (1997)
8. Gnesi, S., Latella, D., Massink, M.: Modular semantics for a UML Statechart diagrams kernel and its extension to Multicharts and branching time model checking. The Journal of Logic and Algebraic Programming **51** (2002) 43–75
9. Lüttgen, G., von der Beeck, M., Cleaveland, R.: A compositional approach to Statecharts semantics. In Rosenblum, D., ed.: Eighth International ACM Symposium on Foundations of Software Engineering, San Diego, California (2000) 120–129
10. Banphawatthanarak, C., Krogh, B.H., Butts, K.: Symbolic verification of executable control specifications. In: Proceedings of the Tenth IEEE International Symposium on Computer Aided Control System Design, Kohala Coast—Island of Hawai'i, HI (1999) 581–586
11. Gargantini, A., Heitmeyer, C.: Using model checking to generate tests from requirements specifications. In Nierstrasz, O., Lemoine, M., eds.: ESEC/FSE '99: Seventh European Software Engineering Conference and Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering. Volume 1687 of Lecture Notes in Computer Science., Toulouse, France, Springer-Verlag (1999) 146–162
12. Rayadurgam, S., Heimdahl, M.P.E.: Coverage based test-case generation using model checkers. In: 8th Annual IEEE Conference and Workshop on the Engineering of Computer Based System (ECBS '01). (2001)
13. Caspi, P., Curic, A., Maignan, A., Sofronis, C., Tripakis, S.: Translating discrete-time Simulink to Lustre. In: Third International ACM Conference on Embedded Software. Volume 2855 of Lecture Notes in Computer Science., Springer-Verlag (2003) 84–99

Improving Use Case Based Requirements Using Formally Grounded Specifications

Christine Choppy[1] and Gianna Reggio[2]

[1] LIPN, Institut Galilée - Université Paris XIII, France
[2] DISI, Università di Genova, Italy

Abstract. Our approach aims at helping to produce adequate requirements, both clear and precise enough so as to provide a sound basis to the overall development. We present a technique for improving use case based requirements, by producing a companion Formally Grounded specification, that results both in an improved *requirements capture*, and in a *requirement validation*. The Formally Grounded requirements specification is written in a "visual" notation, using both diagrams and text, with a formal counterpart (written in the CASL-LTL language). The resulting use case based requirements are of high quality, more systematic, more precise, and its corresponding Formally Grounded specification is available. We illustrate our approach on an Auction System case study.

1 Introduction

While tools and techniques are now available to support quite efficiently software development, one of the most difficult part remains to produce adequate requirements, both clear and precise enough so as to provide a sound basis to the overall development.

Formally based specifications are advocated since they lead to precise, unambiguous descriptions, but they remain difficult to use and impractical in quite a number of cases. We think the reason for this is twofold. One point that was often put forward is the difficulty to write and read such specifications. Another point we see is that it may be difficult to start with formal specifications while still working on the requirements (thus, trying to understand what is the problem about), hence our idea to take advantage of use cases.

Use cases were introduced by Jacobson [10] after the earlier idea of scenarios, which are the different possible courses that different instances of the same use case can take. Use cases are used to describe/capture the requirements of software systems, while providing an overall picture of what is happening in the system. The use case description is textual (it should be "familiar", easy to read) and sums up a set of scenarios.

Use cases are popular because they are easy to use and informal, however "use cases are wonderful but confusing" [7]. A both good and bad thing is that there is a lot of freedom in what should include a use case description, and how it should be written. UML [17] proposes a diagram for use cases, states that descriptions are needed too, and that the sequence of use case activities are documented by behaviour specifications (e.g., with interaction diagrams). However, examples show that use cases are often imprecise, and also that the terms used are vague or ambiguous.

Since use cases are written in the early phases of software development, it is crucial that they should be worked out with a lot of care, so as to avoid to generate errors that will be difficult and costly to correct further on. Interesting work is done to propose some guidelines on how to write use case descriptions, e.g., Cockburn[7] proposes *templates* for structuring their descriptions. In the following, we use an adaptation of this template provided by Sendall[14].

Our idea is to find a way to combine both advantages of use cases and of formal specifications. Here, we present a technique for improving use case based requirements, developing a companion Formally Grounded specification, that results both in an improved *requirements capture* (some requirements may be updated and some may be new), and in a *requirement validation* since writing the specification leads to check that the requirements can be further made explicit up to a precise specification.

The produced requirements specification is written in a "visual" notation, using both diagrams and text, with a formal counterpart which is written in the Casl[11] and Casl-Ltl[12] specification languages.

Being Formally Grounded, our method is systematic, and it yields further questions on the system that will be reflected in the improved use case descriptions. The resulting use case descriptions are of high quality, more systematic, more precise, and their corresponding Formally Grounded specification is available.

In Sect. 2 we shortly sum up our Formally Grounded approach for writing the requirements specification of a software system (see [6] for a full presentation with other examples). In Sect. 3 we present our method for improving use case based requirements using Formally Grounded specifications, and in Sect. 4, we then show how our method applies to a part of the Auction System case study (the complete version is in [5]), showing how the starting use case based requirements have clarified, and how many relevant aspects of the Auction System have been enlightened, before concluding and discussing some related work in Sect. 5.

2 Our Formally Grounded Approach for Requirement Specification

Our Formally Grounded specification approach (see [6] for a complete presentation), aims at helping the user to understand the system to be developed, and to write the corresponding formal specifications. We also support visual presentations of formal specifications, so as to "make the best of both formal and informal worlds". We developed this method for the (logical-algebraic) specification language Casl [11] (Common Algebraic Specification Language, developed within the joint initiative CoFI[1]), and for an extension for dynamic systems

[1] `http://www.cofi.info`

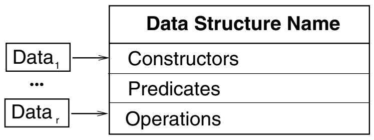
Casl-Ltl[2][12]. Hence, for each visual specification, its formal counterpart in Casl or Casl-Ltl is given.

Our method caters for three different kinds of modelling/specification entities, (i) a data structure, or data type, (ii) a simple dynamic system, that is a single dynamic entity, and (iii) a structured dynamic system, that is composed of mutually interacting dynamic entities; while keeping a common "meta"-structure and way of thinking.

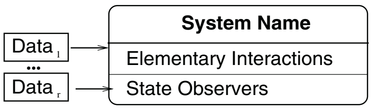
Each entity considered may be modularly decomposed - so its (sub)*parts* are identified-, and is characterized by its *constituent features*. Its model/specification consists of a visual presentation of these parts and constituent features, and of their properties expressed in a natural-language style notation based on an appropriate underlying logic (the variant of logic depends on the kind of entity).

Once the constituent features are identified, we provide guidelines for an exhaustive search of the properties. To this end, we use a tableau whose cells, indexed by the pairs of constituent features, should be filled. For each cell we give a schemata for the relevant properties it should contain, expressing either, when the two indexes are different, the mutual relationships between the two features, or, when they are equal, what is known on that feature. This tableau-filling method ensures that no crucial part of the specification is forgotten, and results in producing a quite structured/navigable set of properties, which should be suitable to support evolution.

Data Structures. Data structures are characterized by a set of values, some constructors to denote those values, and some predicates and operations. Data structures may be structured, e.g., they may import other data structures. These features and the imported data structures (the parts) are visually presented as in the diagram below.
Their properties are expressed in a many-sorted, first order logic [11] with a natural language-like notation. The tableau-filling technique provides a systematic way to find the respective properties of constructors, predicates and operations, e.g., definedness, truth of predicate, etc., see [6].



Data Structure



Simple Dynamic System

Simple Dynamic Systems. Simple dynamic systems are characterized by their states and their transitions, where each transition corresponds to a change of state together with a set of elementary interactions with the external world. Each elementary interaction is described by a name and possibly by parameters (data values). The states are abstractly characterized by "state observers", which, given some parameters, may return some value (operations) or the truth of

[2] LTL stands for Labelled Transition Logic[8,3].

some condition (predicates). Thus, the constituent features of a simple (dynamic) system are its *elementary interactions* with the external world and its *state observers*. The parts of a simple system are its *data structures* needed to define the parameters and the results of elementary interactions and state observers. The above diagram visually presents which are the subparts and the constituent features, while the specification of the parts is given separately. These properties are expressed with a natural language -like notation derived by Casl-Ltl. Casl-Ltl [12] is a Casl extension based on LTL (Labelled Transition Logic) [8,3], a branching time temporal (many-sorted, first-order with edge formulae) logic. This notation uses combinators for expressing that elementary interactions take place (e.g., e **happened**), standard logics (**if - then -else**, **not**, **and**, $=$, **exists**, . . .), and also temporal combinators (**next**, **eventually**, **before**, **in any case**, **in one case**[3]).

A transition from one state to another is characterized by elementary interactions, and properties about states and transitions are expressed, e.g., pre- and post-conditions for elementary interactions, or incompatibilities between them. Properties for a state observer explore e.g., which elementary interactions may cause a change in its value, or which are its possible changes. Again, the tableau-filling technique provides a systematic way to find these properties.

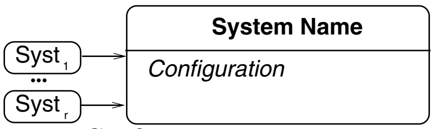
Both diagrams and text have a formal counterpart in the Casl-Ltl language [12].

Structured Dynamic Systems. A structured (dynamic) system is a specialized simple system that is composed of several dynamic systems, its *subsystems*, which can in turn be simple or structured. A transition of such a system should reflect which are the subsystems transitions that occur. Moreover, it is necessary to describe how the subsystems synchronize.

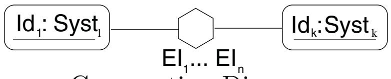
We present here a simpler version of structured systems (the general case is given in [6]) that have only simple subsystems (possibly of different types), where two subsystems may interact only pairwise by performing simultaneously the same elementary interaction, i.e., the behaviour of these structured systems is given by transitions made of groups of subsystem transitions, where each elementary interaction of a subsystem is matched by one of another subsystem. Furthermore, the considered structured systems are closed, i.e., they have no interaction with the external world.

A structured system is visually presented by :
(i) a Context View which is a *configuration diagram* showing the *subsystems* (in the *Configuration*) and their types (the specifications of all those types are given separately), accompanied by a *cooperation diagram* showing the *cooperations* among the subsystems (each cooperation is given by the synchronized execution of elementary interactions, say EI_i).

[3] The last two express universal and existential quantification over execution paths.



Configuration Diagram



Cooperation Diagram

(iii) a Data View which puts together the specifications of all *data structures* that are parts of the system and of its subsystems.

To specify the requirements on a software System it is sufficient to specify a structured dynamic system, whose subsystems are the System itself and all those entities interacting with it (context entities). The specification of the System will be the requirements, whereas the specifications of the context entities will show the assumptions made by the System on the context entities.

3 The Method for High-Quality Requirements

We present in this section our method for producing enhanced requirements. It is organized in five tasks, and works on use case based requirements while developing a companion Formally Grounded specification, which results in improved requirements.

Task 1. Give the use case based requirements on the System following the method proposed by S. Sendall and A. Strohmeier in [13].

Task 2. Find out which are the external entities playing the roles corresponding to the various actors (*context entities*) and determine their types. At this point we can draw a first version of the Context View (see end of previous section), by depicting the System and the found context entities together with their types; the cooperation diagram will have only arcs connecting the System with the context entities.

Task 3. By examining the use case descriptions, one after the other, look for *elementary interactions* and *state observers* of the System; the former should model the interactions between the System and the actors appearing in the use case scenarios, whereas the latter should model information recorded in the System examined or updated in the use case scenarios. Both of them should be depicted in the visual presentation of the System specification (together with the type of their arguments and/or results); the elementary interactions should also be reported in the *cooperation diagram* to show which context entities are taking part in that interactions.

In the meantime put in the Data View any data structure that is used as an argument or a result by a found elementary interaction or state observer. The association between use cases and the elementary interactions and state observers related to it (i.e., which are needed to describe it) should be recorded. During this task, it is possible to find *new entities* interacting with the System that do not correspond to already known actors; they should be added to the Context View, together with their specifications. Whenever, there are relevant assumptions on the context entities they should be made explicit by giving their Formally Grounded specification (they are just simple dynamic systems).

Task 4. Find all the properties about the System following our tableau-filling method. When filling a cell related to some constituent features (elementary interactions and state observers), the descriptions of the associated use cases should be examined as a source of inspiration. During this task, probably, new state observers and data structures will be added, and perhaps the parameters of the existing elementary interactions and state observers may be modified.

Task 5. During the tasks 3 and 4 many *questions* about the System will arise, many aspects of the System that need to be investigated will be highlighted, and many aspects of the System precisely described by the use cases will be found not convincing. These points may be settled following the usual ways, e.g., by interacting with the client, if available, by doing more investigation on the application domain, or by looking at existing similar systems. The produced Formally Grounded specification should reflect the System where all these points have been settled.

The original use case based requirement specification should then be *revised* so as to be coherent with the Formally Grounded one. In general use cases and/ or scenarios may be added or removed, scenarios may be modified by adding/ removing steps or by making more precise the terminology used to describe them. In this way *the final outcome of our method will be not only a better and more systematic understanding of the System reflected in a Formally Grounded specification of the requirements, but also a more precise and sound use case based specification of the same requirements.*

Clearly task 5 will be performed in parallel with tasks 3 and 4.

4 The Auction System Case Study

In this paper we present a part of the application of our method to the case study of an Auction System proposed in [14] by S. Sendall; the remaining parts are in [5]. The description of the problem (from [14]) solved by the Auction System is shown in Fig. 1.

4.1 Auction System Task 1 – Use Case Based Requirement Specification

Here we report the use case based specification of the requirement on the Auction System given following the method of [13] as found in [14]. The only difference with [14] is that we summarize the actors and the use cases by means of a UML use case diagram, see Fig. 2, below, showing also the "*include*" relationships among the use cases (depicted by dotted lines). In the following use case descriptions "**" means that the details about some aspects of the Auction System (e.g., data format or rules to follow to perform some activity) are given in

Your team has been given the responsibility to develop an online auction system that allows people to negotiate over the buying and selling of goods in the form of English-style auctions (over the Internet). The company owners want to rival the Internet auctioning sites, such as, eBay, and uBid. The innovation with this system is that it guarantees bid placed are solvent, making for a more serious marketplace.
All potential users of the system must first enroll with the system; once enrolled they have to log on to the system for each session. Then, they are able to sell, buy, or browse the auctions available on the system. Customers have credit with the system that is used as security on each and every bid. Customers can increase their credit by asking the system to debit a certain amount from their credit card.
A customer that wishes to sell initiates an auction by informing the system of the goods to auction with the minimum bid price and reserve price for the goods, the start period of the auction, and the duration of the auction, e.g., 30 days. The seller has the right to cancel the auction as long as the auction's start date has not been passed, i.e., the auction has not already started.
Customers that wish to follow an auction must first join the auction. Note that it is only possible to join an active auction. Once a customer has joined the auction, (s)he may make a bid, or post a message on the auction's bulletin board (visible to the seller and all customers who are currently participants in the auction). A bid is valid if it is over the minimum bid increment, and if the bidder has sufficient funds, i.e., the customer's credit with the system is at least as high as the sum of all pending bids. Bidders are allowed to place their bids until the auction closes, and place bids across as many auctions as they please. Once an auction closes, the system calculates whether the highest bid meets the reserve price given by the seller, and if so, the system deposits the highest bid price minus the commission taken for the auction service into the credit of the seller (credit internal with the system).
The auction system is highly concurrent–clients bidding against each other in parallel, and a client placing bids at different auctions and increasing his/her credit in parallel.

Fig. 1. Auction System Problem Description

an accompanying document, not present in [14] and thus not considered here. Here we do not detail the schema for the use case description followed in this example and presented in [13]. For lack of room, we present only the main use case buy item under auction, the other ones are in [5].

Use Case buy item under auction

Intention in Context: The intention of the Customer is to follow the auction, which may then evolve into an intention to buy an item by auction, i.e., (s)he may then choose to bid for an item. The Customer may bid in many different auctions at any one time. (Also the actor Participant means the Seller and all the Customers that are joined to the auction).
Primary Actor: Customer
Precondition: The Customer has already identified him/herself to the System.
Main Success Scenario: Customer may leave the auction and come back again later to look at the progress of the auction, without effect on the auction; in this case, the Customer is required to join the auction again.

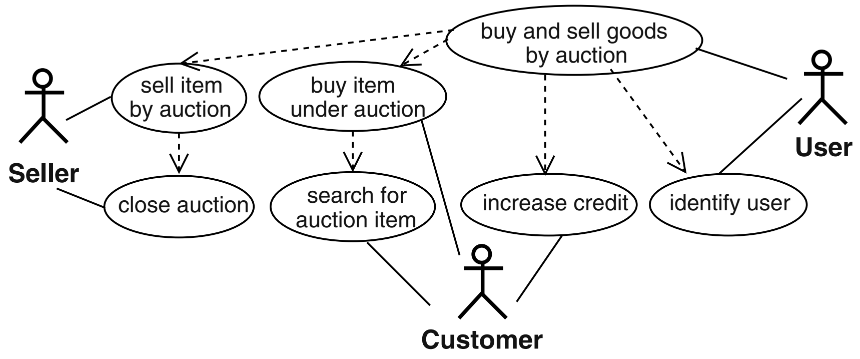


Fig. 2. Auction System: Use cases and actors

1. Customer searches for an item under auction (search item).
2. Customer requests System to join the auction of the item.
3. System presents a view of the auction** to Customer.

Steps 4-5 can be repeated according to the intentions and bidding policy of the Customer

4. Customer makes a bid on the item to System.
5. System validates the bid, records it, secures the bid amount from Customer's credit**, releases the security on the previous high bidder's credit (only when there was a previous standing bid), informs Participants of new high bid, and updates the view of the auction for the item** with new high bid to all Customers that are joined to the auction.

Customer has the high bid for the auction.

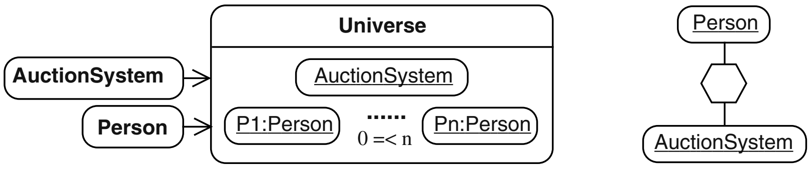
6. System closes the auction with a winning bid by Customer.

Extensions:

2a. Customer requests System not to pursue item further:
  2a.1. System permits Customer to choose another auction, or go back to an earlier point in the selection process; use case continues at step 2.
3a. System informs Customer that auction has not started: use case ends in failure.
3b. System informs Customer that auction is closed: use case ends in failure.
4a. Customer leaves auction:
  4a.1a. System ascertains that Customer has high bid in auction:
    4a.1a.1. System continues auction without effect; use case continues at step 6
  4a.1b. System ascertains that Customer does not have high bid in auction: use case ends in failure.
4||a. Customer requests System to post a message to auction and provides the message content**.
  4||a.1. System informs all Participants of message; use case continues from where it was interrupted.
5a. System determines that bid does not meet the minimum increment**:
  5a.1. System informs Customer; use cases continues at step 4.
5b. System determines that Customer does not have sufficient credit to guarantee bid:
  5b.1. System informs Customer; use cases continues at step 4.
6a. Customer was not the highest bidder:
  6a.1. System closes the auction; use case ends in failure.

4.2 Auction System: Task 2

The Auction System has any number of context entities all of the type Person (anyone accessing the system by Internet). A Person may play three roles: User (plain Internet user), Customer (a User identified by the Auction System and connected with it) and Seller (a Customer selling some goods using the Auction System). We give a first version of the Context View showing the Auction System and the Persons (the incomplete cooperation diagram just shows that the a Person interacts only with the Auction System).



4.3 Auction System: Task 3

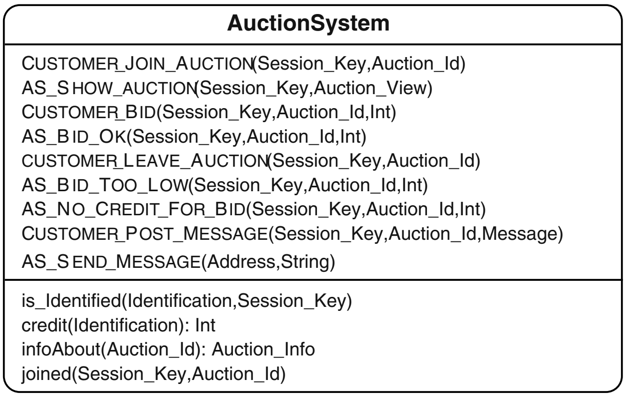
We examine the various use cases, one after the other, looking for the elementary actions and the state observers of AuctionSystem, together with the needed data structures and, possibly new context entities. Note that, for each use case, we do not give the features used by the included sub-use cases.

We name each elementary interaction made by the Auction System with an identifier of the form AS_..., whereas those made by a person context entity will be named USER_..., CUSTOMER_... and SELLER_..., depending on the role.

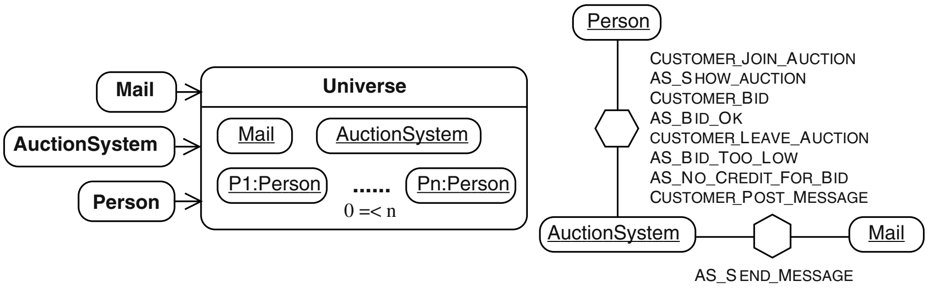
For each use case we produce a fragment of the Context View, of the Data View and of the specification of the Auction System. At the end, all these fragments will be put together getting the initial view of the structural part of the Formally Grounded requirement specification of the Auction System. To be able to support the evolution of the requirements, however, we require to keep track of the features of the specification (elementary interactions, state observers, and data structures) that are related with each use case.

Already, during this task many questions about the Auction System may arise that should be settled with the client; we use the following annotation for these questions and the way chosen to settle them **Q:** *problem* **A:** *settled in this way.*

Use Case buy item under auction. The elementary interactions of AuctionSystem, shown in the first compartment of the above diagram, correspond either to an interaction made in the use case by the Auction System towards a context entity (e.g., AS_BID_OK for communicating that the placed bid was ok) or to an interaction received by a context entity (e.g., CUSTOMER_BID for a Customer placing a bid). Instead, the state observers, in the second compartment, correspond to information recorded inside the Auction System either tested or updated during the use case (e.g., *credit*: the actual credit of a Customer denoted by an identification; *infoAbout*: the current information about an auction).

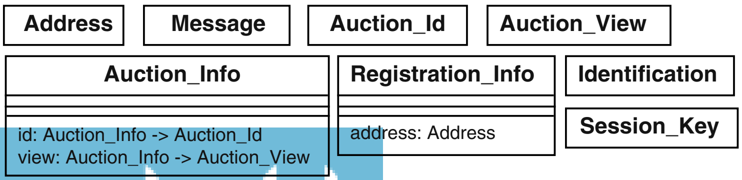

AuctionSystem
CUSTOMER_JOIN_AUCTION(Session_Key,Auction_Id)
AS_SHOW_AUCTION(Session_Key,Auction_View)
CUSTOMER_BID(Session_Key,Auction_Id,Int)
AS_BID_OK(Session_Key,Auction_Id,Int)
CUSTOMER_LEAVE_AUCTION(Session_Key,Auction_Id)
AS_BID_TOO_LOW(Session_Key,Auction_Id,Int)
AS_NO_CREDIT_FOR_BID(Session_Key,Auction_Id,Int)
CUSTOMER_POST_MESSAGE(Session_Key,Auction_Id,Message)
AS_SEND_MESSAGE(Address,String)
is_Identified(Identification,Session_Key)
credit(Identification): Int
infoAbout(Auction_Id): Auction_Info
joined(Session_Key,Auction_Id)

The Context View, see below, shows which context entities take part in the use case (e.g., the person), and which are the interactions of the Auction System with them (e.g., CUSTOMER_JOIN_AUCTION is an interaction between Auction System and the person).



Q: *This use case requires that the Auction System informs the participants to an auction about various facts (e.g., when there is a new higher bid or a message of another participant), but nothing is said on how that will be performed. In the description of the use case* close auction *there is a note saying that this is an open issue and that it will be likely made by email.* **A:** *It is assumed the existence of an external mail service, not further detailed, able to deliver messages to User identified by some kind of address (because the client will decide in future among email, SMS, messaging systems). The mail service will be then a NEW context entity (and a new secondary actor).*

The Data View shows all the data used as parameters by the found elementary interactions and state observers, and which predicates/operations we need to perform all the calculations over them required by the use case. For example, **Auction_Info**, the information about an auction, has an operation, `view`, for recovering a view of the auction to be shown to its participants, whereas **Auction_View** is not further detailed.



4.4 Auction System: Task 4

This task consists in finding the properties about the Auction System by filling the tableau generated by the elementary interactions and state observers found in the previous task, and by completing the specifications of the data structures. Clearly, while doing this activity, new state observers may be added, which will have then to be introduced in the tableaux and considered while looking for the properties. The original use case based specification may be modified by reflecting the better insights on the Auction System gained while looking for properties.

Here we show only some properties, together witht he arisen questions, about a few elementary interactions and state observers needed for the use case buy item under auction; each property is both expressed in our notation, and accompanied by a comment. The full set of the properties can be found in [5].

Elementary Interaction. Customer_Join_Auction Looking for the pre/postconditions of Customer_Join_Auction for filling the tableau cell whose both indexes are that elementary interaction, we found the following unclear points about the Auction System.

Q: *Does the use case* search item *ends having selected one auction or one item? This is relevant because there may be many different auctions for the same item, e.g., a used car. The description of* search item *suggests some auctions, whereas that of* buy item under auction *suggests one item.* **A:** *The* search item *ends with some selected auctions, as in other auction systems.*

Q: *Can an auction selected by the* search item *be in any status (e.g., closed or not yet started)?* **A:** *Yes, and this is quite sensible, since a Customer may be interested in knowing that some item has been sold in the past and at which price, or which are the current starting prices of some items, or that some items will be soon auctioned.*

Q: *Can a Customer try to join a closed or not-started auction?* **A:** *No, the Auction System should not provide this possibility, and answers with an error.*

The above problems lead us to revise the use case search item. As a result, we now have the *NEW* browse auctions use case ending with a selected group of auctions. Moreover, the use case buy item under auction may start only when there is one selected auction that is active. Then, we introduce a new state observer *selected_Auctions* that associates with each identified Customer (referred to by a session key *sk*) the identities of the currently selected auctions.

Q: *Can a Customer join an auction to which (s)he is already joined?* **A:** *Yes, since there is no problem. A better choice may be that the Auction System sends a warning to Customer.*

If a Customer joins an auction, then
  (s)he is identified,
  Customer has selected one auction that is active;
  and after (the Customer has joined that auction, and
  the Auction System shows to her/him all the detail of the selected auction)

```
if CUSTOMER_JOIN_AUCTION($sk$,$aid$) happen then
  exists $id$:$Identification$ s.t. $is\_Identified(id,sk)$ and
    $status(infoAbout(aid)) = active$ and
    $joined^{nxt}(sk,aid)$ and
    in any case next AS_SHOW_AUCTION($sk$,$view(infoAbout(aid))$) happen
```

Elementary Interaction. AS_BID_OK While looking for its postcondition which concerns also the future behaviour of the Auction System after having performed the elementary interaction we detected the following problem.
Q: *Is it true that a Customer joined to an auction is informed twice of each new bid, once by receiving a view of the auction with the new bid and once by some kind of message? Moreover, if a Customer places a bid, and after leaves the auction, will (s)he be ever informed of a new higher bid? More generally, which is the intended duration of an auction? a few hours when the participants bid many times, and continuously look at the current view of the auction? or several days, when the participants from time to time place their bids and look at the situation of the auction?* **A:** *The client decided that an auction handled by the Auction System should last a few hours with all participants logged on; thus there is no need to inform the joined customers and the seller of the various bids, because they continuously examine the current view of the auction that the Auction System keeps updated.*

```
If the Auction System informs a Customer that her/his bid is ok, then
  the Customer placed such bid,
  (s)he had sufficient credit, and the bid met the minimum increment; and after
    the bid is recorded,
    the amount is secured by the Customer credit,
    the security on the previous high bid is released (if any), and
    the updated auction view is sent to all the Customers joined to the auction.
if AS_BID_OK($sk$,$aid$,$i$) happen then
  in any case before CUSTOMER_BID($sk$,$aid$,$i$) happened and
  $i \leq credit(identityOf(sk))$ and
  $ibid\_Ok(infoAbout(aid),i)$ and
  $high\_Bidder(infoAbout^{nxt}(aid)) = identityOf(sk)$ and $high\_Bid(infoAbout^{nxt}(aid))$
  $= i$ and $credit^{nxt}(identityOf(sk)) = credit(identityOf(sk))$ - $i$ and
  (if is defined ($high\_Bidder(infoAbout(aid))$) then
    $credit^{nxt}(high\_Bidder(infoAbout(aid))) =$
    $credit(high\_Bidder(infoAbout(aid))) + high\_Bid(infoAbout(aid)))$ and
  for all $sk_1$:$Session\_Key$
    • if $joined(aid,sk_1)$ then AS_SHOW_AUCTION($sk_1$,$view(infoAbout(aid))$))
```

State Observer. *credit* The first version of the property about the decreasing of the credit (part of the tableau cell indexed by *credit*:*credit*) based on what is written in the various use case descriptions is the following, and points out a problem.

```
If the credit of a Customer decreases, then the Customer made a bid in an auction.
if $credit^{nxt}(id) = credit(id)$ - $i$ and $i > 0$ then exists $sk$:$Session\_Key$, $aid$:$Auction\_Id$
   s.t. AS_BID_OK($sk$,$aid$,$i$) happened and $is\_Identified(id,sk)$
```

Q: *It is true that a Customer using the Auction System only for selling items will be never able to collect her/his money? Moreover, can a buying Customer recover her/his money when (s)he is no more interested in buying?* **A:** *Yes; thus we have to add a NEW use case* decrease credit *for allowing a Customer to recover her/his credit.*

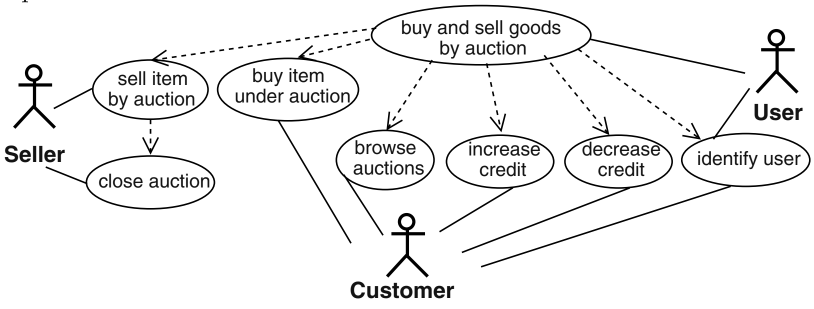
The new version we propose is then

If the credit of a Customer decreases, then
either the Customer asked the Auction System to decrease it, *(NEW)*
or the Customer made a bid in an auction.

```
if $credit^{nxt}(id) = credit(id) - i$ and $i > 0$ then
  exists $sk$:$Session\_Key$, $ctd$ $Credit\_Transfer\_Detail$ s.t.
    AS_DECREASED_CREDIT($sk$,$ctd$) happened and
    $i = amount(ctd)$ and $is\_Identified(id,sk)$
  or exists $sk$:$Session\_Key$, $aid$:$Auction\_Id$ s.t.
    AS_BID_OK($sk$,$aid$,$i$) happened and $is\_Identified(id,sk)$
```

4.5 Auction System Task 5 – New Use Case Based Requirement Specification

Here we report only the new use case diagram and the new description of the use case buy item under auction, see [5] for the complete new use case based requirements. Two new use cases were identified when following our approach (see the previous section), browse auctions (thus, point 1. was removed from the buy item under auction description below) and decrease credit. The questions brought up by our work led to several modifications, e.g., the work on AS_BID_OK in Sect. 4.4 led to remove one part of point 5. in the new buy item under auction description below.



Use Case buy item under auction

Intention in Context: *UNCHANGED*
Primary Actor: Customer
Precondition: The Customer has already identified him/herself to the System
NEW: and selected one active auction.
Main Success Scenario: *UNCHANGED*
*REMOVED: 1. Customer searches for an item under auction (*search item*).*
2. Customer requests System to join the selected auction.

3. System presents a view of the auction** to Customer.
Steps 4-5 can be repeated according to the intentions and bidding policy of the Customer
4. Customer makes a bid on the item to System.
5. System validates the bid, records it, secures the bid amount from Customer's credit**, releases the security on the previous high bidder's credit (only when there was a previous standing bid), *(REMOVED: informs Participants of new high bid,)* and updates the view of the auction for the item** with new high bid to all Customers that are joined to the auction. Customer has the high bid for the auction
6. System closes the auction with a winning bid by Customer.

Extensions:
UNCHANGED: 2a, 5a, 5b, 6a
3a. *NEW: The Customer is the Seller of the auction; System informs Customer that (s)he cannot join the auction. Use case ends with failure.*
REMOVED: 3a. System informs Customer that auction has not started: use case ends in failure.
REMOVED: 3b. System informs Customer that auction is closed: use case ends in failure.
4a. Customer leaves auction:
  4a.1a. System ascertains that Customer has high bid in auction:
    4a.1a.1. System continues auction without effect; use case continues at step 5
  4a.1b. System ascertains that Customer does not have high bid in auction: use case ends in failure.
4||a. Customer requests System to post a message to auction and provides the message content**.
  4||a.1. *MODIFIED: System updates the view of the auction with the added message to all*
  Customers that are joined to the auction; use cases continues from where it was interrupted.

5 Conclusion and Related Works

In this paper we have proposed a method to review use case based requirements for a system by building a companion Formally Grounded specification. As a result the initial requirements are examined in a systematic way through the study of the various aspects of the considered system, modelled in terms of elementary interactions and state observers. For example, the possible interferences among different use cases may be revealed (elementary interactions relative to different use cases may yield a change of the same state observer), the communications between the system and the actors become more precise (they are modelled by elementary interactions, which require a precise definition of their parameters), the secondary actors (that help the system to satisfy the primary actors goals) are discovered and their features are clarified (all entities interacting with the system must be defined and modelled).

The produced Formally Grounded specification has a user-friendly notation (diagrams plus textual annotations in a natural-like language), and so it could be used as the requirement document. The proposed method also requires to update the original use case based requirements whenever a new aspect of the system is brought to light, thus, at the end, new improved use case based requirements are available. In the meantime, the formal Casl/Casl-Ltl specification corresponding to the Formally Grounded one is also available, e.g., for formal analysis (but we have not yet investigated this point).

We think that starting to build directly the Formally Grounded specification from the description of the problem may be not as much as effective as the proposed combination of use cases and Formally Grounded specification, because the ingredients of the Formally Grounded specification (elementary interactions and state observers) are in some sense at a finer grain than the functionalities of the system, and so may be difficult to find by just considering the problem.

As an example, we have used our method on a medium-size case study, an electronic auction system. For lack of room, we have described here only parts of the various tasks and shown only some fragments of the produced artifacts; the complete development and the resulting artifacts can be found in [5]. The advantages shown by our method on this case study seem quite positive. Indeed, we have detected many problematic or not completely clarified aspects in the original use case based requirements. Among them, we recall (i) explicit auctions browsing functionality (blurred in the initial requirements: the information on all auctions were available but not shown), (ii) the fact that the auctions should be performed in a chat-like way, (iii) discovered the need for a decrease-credit functionality, (iv) made explicit that when a Customer unregisters any left credit goes to the Auction System owner.

Moreover, we would like point out that we did not write the starting use case requirements (given by Sendall[14] who, as of now, has no relationship with our group and our method), and we found them quite accurate, presented using a well-organized template and produced following a good method.

Concerning the possibility to use effectively the proposed method we would like to make the following positive points.

– It is possible, using common existing technologies, to build software tools to support the construction of the Formally Grounded specification, not only a graphical editor, but also wizards guiding the properties search.
– Each use case is linked with the elementary interactions, the state observers and the data structures used for its specification. This, together with the precise structure of the properties, may also help to support the evolution of the initial requirements; indeed a modification in one use case may be only reflected in a precise part of the associated Formally Grounded specification.
– The inspection and revision of the requirements proposed by our method concern only the nature of the system to be developed, and does not require to make any choice about the technology and methods that will be used to realize the system; thus it may be used in combination with many different methods.

In the literature there are other approaches to build a formal specification of the requirement of a system, but in general they do not aim at producing an improved non-formal specification. Among them, we recall the nice work of

A. van Lamsweerde and his group[18], which offers a way to formally specify goal-oriented requirement specifications, and then to analyze them by means of formal techniques. R. Dromey[9] proposes to use "Behaviour Tree", a formal-visual notation to specify the requirements, then the resulting requirement specification will be used to derive the architectural structure of the system. Our approach, in the line of the well-founded methods [2], uses the underlying formal foundation to get a rigorous method to precisely specify the requirements, with the aim of achieving a careful inspection and a kind of validation of those requirements.

One of the authors, together with E. Astesiano, proposed another use case based method for the precise specification of the requirements [4], but using the (non-formal) UML statecharts as a notation to describe the use cases. However, because it does not offer a systematic way to analysis the System under different viewpoints, some aspects of the System captured by our method may not come under light.

We would like also to quote the work by S. Sendall and A. Strohmeier [15,16] who promote the use of operation schemas (pre- and postconditions written in OCL) and system interface protocols (UML state diagrams) to complement use cases; our goal is different, that is to improve the use case based requirements.

Inspection techniques for improving the quality of a requirement specification (quite popular in Software Engineering practice, see e.g., [1]) are either based on ad hoc techniques or on check-lists. The main differences with our approach is that our "inspection" based on the underlying formal specification and the tableau-filling technique leads to a more systematic and precise examination of the requirements, whereas standard techniques lead to more generic checking. For instance, compare:
"find and list all the ways the *credit* state observer may be updated in the various scenarios of all use case" (which helped to discover the lacking functionality of credit decreasing),
with
"Is there any missing functionality, that is, do the actors have goals that must be fulfilled, but that have not been described in use cases?" taken from [1]'s check-list.

References

1. B. Anda and D. Sjoberg. Towards an Inspection Technique for Use Case Models. In *Proc. SEKE 2002*. ACM Press, 2002.
2. E. Astesiano, G. Reggio, and M. Cerioli. From Formal Techniques to Well-Founded Software Development Methods. In *Proc. of The 10th Anniversary Colloquium of the United Nations University International Institute for Software Technology (UNU/IIST): Formal Methods at the Crossroads from Panacea to Foundational Support. Lisbon - Portugal, March 18-21, 2002.*, Lecture Notes in Computer Science. Springer Verlag, Berlin, 2003. To appear. Available at `ftp://ftp.disi.unige.it/person/ReggioG/AstesianoEtAll03a.ps`, and `ftp://ftp.disi.unige.it/person/ReggioG/AstesianoEtAll03a.pdf`.
3. E. Astesiano and G. Reggio. Labelled Transition Logic: An Outline. *Acta Informatica*, 37(11-12):831–879, 2001.

4. E. Astesiano and G. Reggio. Tight Structuring for Precise UML-based Requirement Specifications. In *Proc. of Monterey Workshop 2002: Radical Innovations of Software and Systems Engineering in the Future. Venice - Italy, October 7-11, 2002.*, Lecture Notes in Computer Science. Springer Verlag, Berlin, 2003. To appear. Available at `ftp://ftp.disi.unige.it/person/ReggioG/AstesianoEtAll03f.pdf`.
5. C. Choppy and G. Reggio. Improving Use Case Based Requirements Using Formally Grounded Specifications (Complete Version). Technical Report DISI-TR-03-45, DISI – Università di Genova, Italy, 2003. Available at `ftp://ftp.disi.unige.it/person/ReggioG/ChoppyReggio03c.ps`, and `ftp://ftp.disi.unige.it/person/ReggioG/ChoppyReggio03c.pdf`.
6. C. Choppy and G. Reggio. Towards a Formally Grounded Software Development Method. Technical Report DISI–TR–03–35, DISI, Università di Genova, Italy, 2003. Available at `ftp://ftp.disi.unige.it/person/ReggioG/ChoppyReggio03a.pdf`.
7. A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2000.
8. G. Costa and G. Reggio. Specification of Abstract Dynamic Data Types: A Temporal Logic Approach. *T.C.S.*, 173(2):513–554, 1997.
9. R. Dromey. From Requirements to Design: Formalizing the Key Steps. In *Proc. of SEFM'03, Brisbane - Australia*. IEEE Computer Society, 2003.
10. I. Jacobson, M. Christerson, P. Jonnson, and G. Overgaard. *Object-Oriented Software Engineering: A Use-Case Driven Approach*. Addison-Wesley, 1992.
11. P. Mosses, editor. *CASL, The Common Algebraic Specification Language - Reference Manual*. Lecture Notes in Computer Science. Springer-Verlag, 2003. To appear. Available at `http://www.cofi.info/CASL_RefManual_DRAFT.pdf`.
12. G. Reggio, E. Astesiano, and C. Choppy. CASL-LTL : A CASL Extension for Dynamic Reactive Systems Version 1.0– Summary. Technical Report DISI-TR-03-36, DISI – Università di Genova, Italy, 2003. Available at `ftp://ftp.disi.unige.it/person/ReggioG/ReggioEtAll03b.ps`, and `ftp://ftp.disi.unige.it/person/ReggioG/ReggioEtAll03b.pdf`.
13. S. Sendall and A.Strohmeier. Requirements Analysis with Use Cases. `http://lglwww.epfl.ch/research/use_cases/RE-A2-theory.pdf`, 2001.
14. S. Sendall. Case studies for RE_A2 course "Requirements Analysis with Use Cases". `http://lglwww.epfl.ch/research/use_cases/RE-A2-case-studies/index.html`, 2001.
15. S.Sendall and A.Strohmeier. From Use Cases to System Operation Specifications. In S. K. A. Evans and B. Selic, editors, *Proc. UML'2000*, number 1939 in Lecture Notes in Computer Science, pages 1–15. Springer Verlag, Berlin, 2000.
16. S.Sendall and A.Strohmeier. Specifying Concurrent System Behavior and Timing constraints using OCL and UML. In M. Gogolla and C. Kobryn, editors, *Proc. UML'2001*, number 2185 in Lecture Notes in Computer Science, pages 391–405. Springer Verlag, Berlin, 2001.
17. UML Revision Task Force. *OMG UML Specification 1.3*, 2000. Available at `http://www.omg.org/docs/formal/00-03-01.pdf`.
18. A. van Lamsweerde. Building Formal Requirements Models for Reliable Software (Invited paper). In *6th International Conference on Reliable Software Technologies, Ada-Europe 2001*, number 2043 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2001.

The GOPCSD Tool: An Integrated Development Environment for Process Control Requirements and Design

Islam A.M. El-Maddah and Tom S.E. Maibaum

Department of Computer Science, King’s College London
London WC2R 2LS, UK
{elmaddah,tom}@dcs.kcl.ac.uk
Fax +4402078482851

Abstract: The GOPCSD (Goal Oriented Process Control Systems Design) tool is an integrated environment, where the process control systems engineer can construct, import, check, reason about, modify, validate requirements specifications and generate in the B specification language a formal specification of such process control requirements. Borrowing from the KAOS method, the GOPCSD tool adopts the goal-oriented hierarchy concept to enable easy tracing of the user needs to the requirements level, as well as the requirements to the design specification level. The tool offers a library and formal and informal checks and tests to aid correction and enhancement of the requirements; in addition, the normal systems engineer can use the tool effectively and automatically generate a B formal specification, thus not demanding a high-level of knowledge about the sophisticated mathematics supporting formal methods like B.

1 Introduction

In reactive systems, such as process control systems, where safety and security are considered important aspects, along with the system’s operational constraints, the B formal method has been demonstrated to provide effective support at the early design stages. However, a requirements stage before the formal method is still needed, concerned with the construction, reuse, correction, modification, testing and enhancement of the requirements.

In addition, tracing the user needs to the requirements level, as well as tracing each requirement to the design specification level decreases the gap between the user’s perspective and the specification level. This motivated us to adopt the goal-oriented requirements analysis method of KAOS [3] as a starting point. The goal driven requirements analysis method of KAOS uses goal-models as the formal model to structure the software requirements gradually and precisely; the main goals of the goal-model usually represent the user needs, while the low-level goals will be translated to specification building blocks. This ensures the traceability goal as well as the understandability of the goal-model by the user.

Thus, we were motivated to start the requirements analysis as close to the view of the systems engineer as in [8] and, furthermore, to extend the formalisation, via an automated tool, to the B specification level [1], as in [6, 9]. In this paper, we introduce the GOPCSD tool that serves as a front end for the B toolkit [7] or other environments that adopt B, to hide the mathematical details of the B method and to allow the systems engineer to focus on providing precise and formal requirements, while preparing the stage for the software engineer to use the generated B specification to produce code that is not only correct with respect to the high-level B specifications, but is also much more likely to satisfy the systems engineer's stated requirements.

2 The GOPCSD Tool

The GOPCSD tool [2] (as shown in fig.1) is designed to analyze the requirements of process control systems and automatically generate B formal specifications. To build an application within the tool, there are three main phases. In the first phase, the requirements can be constructed using different (process control specific) entities (components, agents, variables, goals, goal-models, and variable types). The second phase checks the consistency, completeness, reachability, and the validity of the goal-model. The tool suggests different goal-model modifications to resolve goal-conflict or unreachable goals, avoid obstacles, or complete the requirements. Thus, the first and second phases can be regarded as a feedback loop to devise consistent and complete requirements. Finally, after the goal-model is considered satisfactory, the application proceeds to the third and final phase producing automatically the software specification through translating the goal-model to a B specification. The GOPCSD tool hides the details of the B language from the systems engineer, and increases the separation of concerns between the software and systems engineers.

Although the GOPCSD method is based on KAOS, there are some significant adaptations of the KAOS method to fit with the nature of Process Control Systems and to enable an automatic phase to generate B specifications to be added. The tool restricts the variables of the application to have finite domain in order to enable feedback for the completeness, consistency and validity of the requirements.

2.1 GOPCSD Tool Support for Reusability

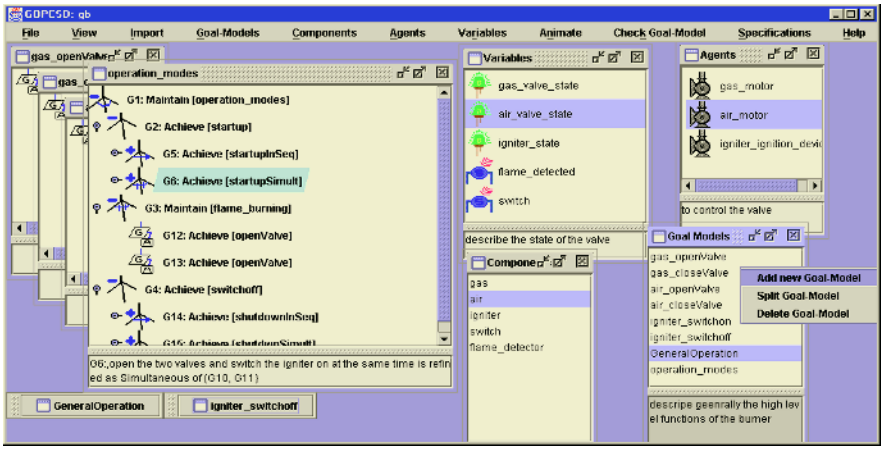
Since the main target of the GOPCSD tool is process control systems, this motivates a need for building systems from sub-systems and components. Therefore, the GOPCSD tool supports reusability in building the goal-models based on two basic concepts: similar applications are built of the same kinds of physical components, and similar applications can have similar high-level goals, even if they have different components.

The first concept suggests a library for the frequently used components with their associated low-level goal-models to enable the user to import them into his/her applications. The second concept recommends high-level goal-model templates that can be used in similar systems with possibly different component details. For example, a production cell with a single press can have some of the same high-level goals as a double-press production cell.

2.2 Constructing Goal-Models (Phase I)

The GOPCSD method mainly depends on reusing elements of the library to increase the reusability and the maintainability of the developed applications. Thus, the design starts by importing components from the library in addition to the high-level goal-model templates. After the systems engineer decides on the components and/or the high-level goal-model templates to be imported, a re-naming and/or mapping process can take place in order to change the components' or the templates' variables and/or agents' names into the corresponding ones to be used in the new application.

Then, the systems engineer formulates the application's main goals; these generally can be considered as medium-level goals between the high-level goal-model templates and the low-level component goal-models. The main goals can be placed in separate empty goal-models until the user links them by combining them into a higher-level goal. The process of producing a single complete goal-model requires the user to link the high-level goal models, the main goals and the low-level goal-models of the components.



G1: Maintain [operation_modes]
G2: Achieve [startup]
G5: Achieve [startupInSeq]
G6: Achieve [startupSimult]
G3: Maintain [flame_burning]
G12: Achieve [openValve]
G13: Achieve [openValve]
G4: Achieve [switchoff]
G14: Achieve [shutdownInSeq]
G15: Achieve [shutdownSimult]

G6:,open the two valves and switch the igniter on at the same time is refined as Simultaneous of (G10, G11)

gas_valve_state
air_valve_state
igniter_state
flame_detected
switch

describe the state of the valve

gas_motor
air_motor
igniler_ignilion_devic

to control the valve

gas
air
igniter
switch
flame_detector

gas_openValve
gas_closeValve
air_openValve
air_closeValve
igniter_switchon
igniter_switchoff
GeneralOperation
operation_modes

descripe geenrally the high level functions of the burner

Fig. 1. The desktop of the GOPCSD tool, showing the requirements elements (components, variables, agents and goal-models lists) to the right and a goal-model to the left.

2.3 Checking the Goal-Model (Phase II)

The GOPCSD tool enables the user to perform various checks and tests to enhance the requirements and to detect any incorrect or undesirable behaviours implied by the requirements, as early as possible. The tool offers the following checks:

Checking Correctness of Goal-Models. There are some essential constraints the tool enforces on the goal-models in order to build correct B specification (and correctness is defined with respect to these constraints and those pertaining to the structure of goal trees). These constraints address the basic definitions of terminal goals, variable controllability, and refinement patterns. This utility will highlight the goals that violate the constraints.

Reasoning and Investigation Utilities. The GOPCSD tool enables the user to reason about the how and why of the different goals of the goal model. Reasoning how to achieve one goal lists the sub-goals of this goal and their sub-goals; while reasoning why to achieve one goal ascends the goal-model level by level listing the details of the ancestor goals. The reasoning utility can be regarded as an informal early level of checking to validate parts of the goal-model, on the one hand; on the other, it can be used to guide the user to elicit new goals to complete the construction of the goal-model, either upward answering why or downward answering how [3]. Another utility offered by the tool is highlighting the goals containing a specific variable or the goals that are affected by a specific agent. This utility is similar to dependence graphs, or tracing utilities, which can be helpful to judge variable and agent coupling.

Checking the Reachability of the Requirements. Another important check that is helpful to amend the requirements is detecting unreachable goals. The user usually errs in specifying the conditions of some of the goals or locates them in inappropriate positions where they will never be activated. Thus, the goal-model can be valid, complete and consistent but some of its goals' pre-conditions can be unreachable.

Checking Completeness. Completeness means that for each combination of the application's variables there is a defined action(s) to be taken. Some incompleteness can occur as a result of ignoring unexpected variable combinations or ignoring variable combinations under specific situations. The systems engineer may choose not to include such combinations for normal operation, but during the application execution they might occur and produce a hazard or incorrect operation.

Checking Obstacles. An obstacle is a sequence of events that can occur during application run-time and can obstruct some of the goals from being achieved [4]. Obstacle analysis can be performed at the lowest level within the complete goal-model, when all the terminal goals have a complete formal description. Each formal description of a terminal goal will be negated in turn and the user attempts to find cases that validate the negated conditions and, hence, the goal-model can be modified in order to prevent the obstacles from occurring or to attenuate their effects.

Checking Goal Conflict. Conflict arises when two or more goals prescribe the performance of inconsistent actions under the same conditions [5]. Conflicts are checked by comparing the conditions for goals that assign different values to the same variables. When a goal-conflict is detected, the tool suggests that one of the conflicting goal's formal pre-conditions should be modified. This utility can be employed to bring together the various process control requirements aspects, such as safety, security and productivity. In case of conflicts, the GOPCSD tool will suggest to modify the operational goals in order to pay attention to the various issues.

Animating Goal-Models. The systems engineer usually needs to validate the requirements. Such validation should emulate the execution of the controller during run-time. The GOPCSD provides different utilities, such as saving, loading and executing event lists, emulating output delay and faults, and displaying the description of the activated goals, cycle by cycle. This validation utility provides the user with an easy to use means of understanding the requirements model. In addition, it guides the user to fine-tune the goal-model of the application to enhance the requirements and remove bugs as early as possible.

3 Translating Goal-Models to B Machines (Phase III)

After the user validates the goal-model and approves the description of the controller, the tool translates the corrected requirements into formal specifications represented as B machines. The translation from goal-models to B specifications consists of two steps, as follows:

3.1 Splitting Compound Goal-Models

Alternative refinement is used to enable the user to express more than one alternative solution for the control application. To reduce the effort expected form/by the user, the shared parts of the compound goal-model are available for each version. However, to build controller specifications, only a single solution can be used. Thus, the translation into B specifications starts by splitting the goal model if it has alternative refinement sites.

3.2 Translating Goal-Models to the B AMN Language

The complete goal-model, which represents a separate solution, will be automatically translated by the tool to a main controller B machine and a number of actuator B machines, depending on the number of the active agents used within the goal-model. In addition, the definitions of the various data types of the variables will be collected in a data type B machine. The different values that can be assigned to the output variables will be represented as operations of the actuator machines. The terminal goals of the goal-model will be grouped by the output variable they control and will be translated to parallel parts of one operation of the main controller machine; this will ensure that the controller can manipulate the output variables in parallel.

References

1. J. R. Abrial, The B Book: Assigning Programs to Meaning, Cambridge University Press, 1995.
2. I. A. El-Maddah and T. S. Maibaum, Goal-oriented Requirements Analysis of Process Control Systems, first MEMOCODE, France, 2003

3. A. Van Lamsweerde, A. Dardenne, B. Delcourt, and F. Dubisy, The KAOS Project: Knowledge acquisition in automated specifications of software, proceeding AAAI Spring Symposium series, Track: Design of Composite Systems, Stanford University, March 1991, pp 59-62.
4. A. Van Lamsweerde and E. Letier, Obstacles in Goal-driven Requirement Engineering, proceeding ICSE'98 20th international conference on software engineering, Kyoto, ACM-IEEE, April 1998.
5. A. Van Lamsweerde, R. Darimont and E. Letier, Managing Conflicts in Goal-Driven Requirement Engineering, IEEE Transactions on Software Engineering, Special Issue on Managing Inconsistency in Software Development, Nov. 1998.
6. K. Lano, K. Androutsopoulos, and D. Clark, Structuring and Design of Reactive Systems using RSDS and B, FASE, ETAPS 2000
7. K. Lano and H. Haughton, Specifications in B, An introduction using the B toolkit, 1996, Imperial College Press
8. E. Letier and A. V. Lamsweerde, Deriving Operational Software Specifications from System Goals, SIGSOFT 2002/FSE-10, Nov. 18-22 Charleston, SC, USA.
9. E. Sekerinski, Graphical Design of Reactive Systems, D. Bert (Ed.) 2nd International B conference, Montpellier, France, Springer-Verlag, 1998.

Automated Debugging Using Path-Based Weakest Preconditions

Haifeng He and Neelam Gupta

Dept. of Computer Science, The University of Arizona, Tucson, Arizona 85721
{hehf,ngupta}@cs.arizona.edu

Abstract. Software debugging is the activity of locating and correcting erroneous statements in programs. Automated tools to locate and correct the erroneous statements in a program can significantly reduce the cost of software development. In this paper, we present a new approach to locate and correct an erroneous statement in a function. We assume the correct specification of the erroneous function is available in the form of preconditions and postconditions of the function. Our approach combines ideas from software testing and weakest preconditions used in correctness proof methods to locate a likely erroneous statement. We have implemented our approach and conducted experiments with several small programs. In our experiments, our approach was able to locate the erroneous statements in a large number of cases. Our preliminary experimental results show that our approach has potential for development of an automated bug location and correction tool.

Keywords: Fault location, software testing, weakest precondition, postcondition.

1 Introduction

Software debugging is the process of locating and correcting erroneous statements in a faulty program. It is an expensive and challenging activity requiring understanding of the program and is often done manually by the programmers. Automated tools that help the programmers in locating the erroneous statements can significantly reduce the cost of software development.

The program slicing based approaches [1,11,15] extract a subset of program statements that can effect the values of variables at the point where a fault is manifested. A novel approach to automatically isolate cause-effect chains, that have higher precision than static or dynamic slices, has been developed in [19,9]. Approaches based on dynamic invariant detection [8,16] to give warnings about program anomalies have also been developed. All these approaches assist the programmers by narrowing down the search for erroneous statements to a subset of program statements. However, they do not generate the exact modifications to be made to the program to automatically correct the errors. To determine the exact nature of an error and check whether it lies in the localized program statements, the programmers have to modify the program and re-execute the program until they obtain correct output.

In this paper, we develop an approach to automatically *localize and correct* an erroneous statement in a faulty function. Our approach assumes that the precondition and the postcondition of the function are available as first order theory formulas (FOT) [6] formulas over a finite domain. We also assume that a test suite for a given test adequacy criteria for structural testing of the given function is available. In our current work, we also assume that there is at most one error statement in the faulty function to avoid interaction among multiple errors. Our method takes as input an error trace generated by executing some failed test case in the test suite of the function. The notion of weakest precondition [5,7] of a statement in a program for a given postcondition of the program has been used for proving program correctness. In this paper, we define a notion of *path-based weakest precondition* for statements along a path in a program. Using this, we also define the notions of a *hypothesized program state* and an *actual program state* at every point along the error trace. Our algorithm traverses the statements along the trace in reverse order of execution and compares these states at each point along the trace to detect an *evidence* for a likely faulty statement. It then generates modifications to the function to remove this evidence. The algorithm terminates if a modified function successfully executes all the test cases in the test suite. If all the statements along the current error trace have been processed and the algorithm fails to correct the error, another error trace (corresponding to another failed test case in the test suite) is tried until all the error traces have been attempted.

We have implemented our algorithm and conducted experiments with several small programs by introducing one error at a time. In our experiments, our technique was able to correct errors such as a wrong relational operator used in a branch predicate, wrong variable used in a branch predicate, wrong variable used in an assignment statement, incorrect constant used in an assignment statement and some cases of incorrect number of loop iterations. Our approach requires normal *termination* of the program execution for the input that generated the error trace so that the postcondition can be evaluated for this input. Therefore, it is not able to correct errors that result in a non-terminating loop or result in segmentation faults such as due to illegal memory access.

The organization of this paper is as follows. The terminology used and the details of our approach are explained in section 2. The steps of our algorithm are described in section 3. The experiments are presented in section 4, the related work is discussed in section 5 and conclusions are mentioned in section 6.

2 Our Approach

The problem addressed in this paper can be stated as follows:

Problem Statement: *Given a faulty function F with a single error statement, a test suite T_s for F, an error trace of F generated by T_s, and the precondition and postcondition of F in the first order predicate logic, localize and modify a statement in the error trace so that the modified function is able to pass the test cases in T_s.*

First, we execute F with T_s and identify the set of error traces i.e., the set of execution traces for which the postcondition of F evaluates to *False*. We select any one of these error traces for locating and correcting the error in the function. We also express the postcondition in disjunctive normal form using the input for the trace. Having the postcondition in disjunctive normal form, we only need to guarantee the validity of one conjunction in the postcondition to satisfy the postcondition. We select any one of the conjunctions in the evaluated postcondition for the error trace for locating and correcting the error in the error trace and call it *postcondition conjunction* R. If the algorithm is unable to fix the error for given trace with the selected postcondition conjunction, another conjunction in the postcondition conjunction is selected. The steps of our algorithm are shown in 2. Next, we describe our representation of an error trace. We illustrate our approach with a faulty program *Max* to compute a maximum element in an unsorted array of integers, shown in Figure 1(a). In this program, incorrect relational operator is used in line 4. An error trace with the input a[]= (-2, 5, 3), n=3 is shown in Figure 1(b).

```
int Max(int a[], int n){
int i, s;
precondition n>0
1:   i = 1;
2:   s = a[0];
3:   while (i < n) {
4:      if (s ≥ a[i])
5:         s = a[i];
6:      i = i + 1;
     }
     return s;
     postcondition
     ∀i:0≤i<n, s≥a[i] ∧
     ∃i:0≤i<n, s=a[i]
}
```

precondition (n>0)

$<i, line\#>$	$Stmt_i$
< 1,1>	`i=1`
<2,2>	`s=a[0]`
<3,3>	`(i-n<0)`
<4,4 >	`(s-a[i=1]<0)`
<5,6>	`i=i+1`
<6,3>	`(i-n<0)`
<7,4>	`(s-a[i=2]<0)`
<8,6>	`i=i+1`
<9,3>	`(i-n≥0)`

postcondition

$((s\text{-}a[0]\geq 0)^T \wedge (s\text{-}a[1]\geq 0)^F \wedge (s\text{-}a[2] \geq 0)^F \wedge (s\text{-}a[0]=0^T)\ \vee$
$((s\text{-}a[0] \geq 0)^T \wedge (s\text{-}a[1]\geq 0)^F \wedge (s\text{-}a[2]\geq 0)^F \wedge (s\text{-}a[1]=0^F)\ \vee$
$((s\text{-}a[0]\geq 0)^T \wedge (s\text{-}a[1]\geq 0)^F \wedge (s\text{-}a[2]\geq 0)^F \wedge (s\text{-}a[2]=0)^F)$

Fig. 1. (a) A Program `Max` 1.(b) A trace of `Max` with the input a[]=(-2, 5, 3), n=3.

2.1 Representation of an Error Trace

An error trace is the execution history of a failed test case. We define it as a sequence of executed instances of statements (assignments, branch predicates, input/output statements) and evaluated precondition and postcondition of the function. We define an *execution point* i, $(i = 1, n)$ in a trace as the entry of the i^{th} statement instance executed in the above sequence. We use *bottom* to denote the exit point of the last (n^{th}) statement in the trace.

We use a tuple $< i, j >$ to indicate an instance of an executed statement in an error trace, where i is the execution point of the statement instance in the error trace and j is the line number of the statement in the program. For simplicity, we assume that there is only one statement at a line in the program. Since the execution point of a statement instance is unique in a trace, we denote a statement instance at $< i, j >$ as $Stmt_i$. An error trace generated by executing the function in Figure 1(a) with the input a[]=$(-2, 5, 3)$, n=3 is shown in Figure 1(b).

Representation of Branch Predicates. We define an *atomic predicate* as having the form $(expr\ relop\ const)$, where $expr$ is an arithmetic expression without a constant term, $relop$ is a relational operator $(<, \leq, >, \geq, =, \neq)$ and $const$ is a constant term. At present, we have considered the branch predicates that use only real, integer and character data types. The branch predicates along the trace are represented in the above form. For example, the `while` predicate `(i<n)` at line 3 in Figure 1(a) is formalized as $Stmt_3$:`(i-n<0)` in the error trace in Figure 1(b). The compound predicates are represented in *disjunctive normal form* i.e., one that has the form $E = e_0 \vee \cdot\cdot\cdot \vee e_n$, where each e_i has the form $g_0 \wedge \cdot\cdot\cdot \wedge g_m$ and each predicate g_i is an atomic predicate represented in the above standard form.

During program execution, the values of the array indices are known. We denote an array element in the trace as `array[idx = const]`, where `array` is the name of the array, `idx` is the expression for the index of the array element in the program, and `const` is the value of `idx` for the input used for the trace. In addition, if a branch predicate evaluates to true, then it is shown in the trace as it is; otherwise, its negation, which must be true, is shown in the trace. For example, let us consider the control statement `if (s` $\geq$ `a[i])` at line 4 in the example program in Figure 1(a). This statement is executed at position 7 in the trace in Figure 1(b). At that point, the value of `(s` $\geq$ `a[i])` is *False*. Thus, its negation is shown in the trace in Figure 1(b). The corresponding statement instance in the trace is $Stmt_7$: `(s-a[i=1]<0)`. Next, we describe our representation of precondition and postcondition of a program.

Representation of Precondition and Postcondition. Using the program input for the error trace, we transform the precondition and postcondition of the program into the disjunctive normal form. The transformation is done in two steps. First, during the execution, the precondition and postcondition are transformed into *quantifier-free* predicates. Using the program input for the error trace, the universal quantifier $\forall$ is expanded as a conjunction and the existential quantifier $\exists$ is replaced with a disjunction. For example, for $n = 3$, the quantifier $\forall\ i:\ \ 0 \leq i < n, s \geq a[i]$ in the postcondition of the program in Figure 1(a) is expanded as $(s \geq a[0]) \wedge (s \geq a[1]) \wedge (s \geq a[2])$. An existential quantifier is expanded as a disjunction. For example, for $n = 3$, the quantifier $\exists\ i:\ 0 \leq i < n, s = a[i]$ in the postcondition of the program in Figure 1(a) is expanded as $(s = a[0]) \vee (s = a[1]) \vee (s = a[2])$. The second step is to convert *quantifier-free* precondition or postcondition into the disjunctive normal. For the trace in Figure 1(b), we obtain the postcondition in the disjunctive normal from as below.

$$\begin{array}{l}((s-a[0] \geq 0) \wedge (s-a[1] \geq 0) \wedge (s-a[2] \geq 0) \wedge (s-a[0] = 0))\vee \\ ((s-a[0] \geq 0) \wedge (s-a[1] \geq 0) \wedge (s-a[2] \geq 0) \wedge (s-a[1] = 0))\vee \\ ((s-a[0] \geq 0) \wedge (s-a[1] \geq 0) \wedge (s-a[2] \geq 0) \wedge (s-a[2] = 0))\end{array}$$

We classify a predicate that evaluates to true with the given input as a *positive predicate* and a predicate that evaluates to False with the given input as a *negative predicate*. We use a superscript on each atomic predicate in the postcondition to show the truth value of that predicate for the given input. For example, in the trace in Figure 1(a) $(s - a[0] \geq 0)^T$ means that this predicate evaluates to true and $(s - a[1] \geq 0)^F$ means that it evaluates to False. Note that all the branch predicates in the trace after their representation in the standard from are positive predicates.

2.2 Weakest Precondition and Path-Oriented Weakest Precondition

Given a program S and the postcondition R, the **weakest precondition wp(**S**,** R**)** represents the set of all states such that the execution of S begun in any of them is guaranteed to terminate in a state satisfying R [5,7].

In this paper, we define a weakest precondition semantics with respect to a trace, which we call as **path-based weakest precondition**, or **pwp** for abbreviation.

Definition: Given an execution trace T and the postcondition R of a function F, the **path-based weakest precondition** denoted as **pwp(**T**,** R**)** is the set of all states such that an execution of F, that follows T, begun in any of them is guaranteed to terminate in a state satisfying R.

The control flow in a trace is fixed so only the data dependences affect the value of output. Assume that evaluation of control statements does not have any side effects, we formally define **pwp** as below.

pwp$(x = a, R) = R_{x \to a}$, where $x \to a$ means substituting every occurrence of x in R with a

pwp$(B, R) = R$, where B is a branch predicate

pwp$(C; D, R) =$ **pwp**$(C,$ **pwp**$(D, R))$

Given a subtrace $T_{<i,n>}$ of T (from execution point i to the end of trace) and a postcondition R, we denote **pwp(**$T_{<i,n>}, R$**)** as R_i. For example, let us consider the trace in Figure 1(b) and let assume that the postcondition conjunction R be

$((s - a[0] \geq 0) \wedge (s - a[1] \geq 0) \wedge (s - a[2] \geq 0) \wedge (s - a[0] = 0))$. The path-based weakest precondition at different execution points along the trace is:

$R_3 = R_4 = R_5 = R_6 = R_7 = R_8 = R_9 = R_{bottom} = R$

$=((s - a[0] \geq 0) \wedge (s - a[1] \geq 0) \wedge (s - a[2] \geq 0) \wedge (s - a[0] = 0))$ $(s - a[1] \geq 0)$.

$R_1 = R_2 =$ **pwp**(s=a[0], R_3) = $((0 \geq 0) \wedge (a[0] - a[1] \geq 0) \wedge (a[0] - a[2] \geq 0) \wedge (0 = 0))$.

As seen in this example, in order to compute R_i at each point in a trace, we only need to know the set of the assignment statements that are needed for computation of R_i.

Definition: Given a trace T and the postcondition R of a function F, the **pwpSlice** $S(T, R, i)$ is an ordered set of assignment statements from point i to the end of T, upon which the value of R is directly or indirectly data dependent.

In other words, the **pwpSlice** $S(T, R, i)$ consists of all the assignment statements that are needed for computation of R_i. In the above example, $S(T, R, 1)$ is $\{Stmt_2\}$. At each execution point i on an error trace, we compare the atomic predicates in the predicate representing the set of *hypothesized program states* and the predicate representing the set of *actual program states* to look for an evidence for locating the error in the trace.

2.3 Hypothesized Program State

The set of *hypothesized program states* at an execution point along the trace is represented by a predicate in disjunctive normal form derived from the postcondition as explained below.

Definition: Given a trace T and a postcondition R of a function F, the set of **hypothesized program states** at an execution point i along the trace is defined as the path-based weakest precondition R_i = **pwp(**$T_{<i,n>}, R$**)**.

The set of hypothesized program states R_i at any execution point i, (i=1, n) along the trace is computed as R_i = **pwp(**$Stmt_i, R_{i+1}$) for i=1, n-1 and R_n = **pwp(**$Stmt_n, R$**)**.

2.4 Actual Program State

The set of *actual program states* at an execution point along a trace is represented by a predicate in disjunctive normal form that is actually true for the given input. It consists of a set of *forward program states*, Q_i^F, and a set of *backward program states*, Q_i^B. The set of forward program states Q_i^F at an execution point i along a trace T is defined as:

Q_1^F = positive conjunctions in precondition.
Q_i^F = (Q_{i-1}^F - $Kill_{i-1}$) $\cup$ Gen_{i-1}, i=1,n
Q_{bottom}^F = (Q_n^F - $Kill_n$) $\cup$ Gen_n, i=1,n

where $Kill_{i-1}$ is the set of predicates killed by statement instance $Stmt_{i-1}$ and Gen_{i-1} is the set of predicates derived from $Stmt_{i-1}$. A predicate p is killed by $Stmt_{i-1}$ if there is a variable in p that is defined at $Stmt_{i-1}$. For example, `(i-n<0)` is killed by statement `i=i+1`. Since i is redefined, after `i=i+1` is executed, `(i-n<0)` may not hold. If $Stmt_{i-1}$ is an assignment statement, then an equivalence is derived from $Stmt_{i-1}$. If $Stmt_{i-1}$ is a branch predicate, then Gen_{i-1} is the set of predicates in $Stmt_i$. The computation of the set of forward program states Q_4^F for the error trace in Figure 1(b) is shown below.

Q_1^F = (n>0)
Q_2^F = (n>0)$\wedge$(i=1)
Q_3^F = (n>0)$\wedge$(i=1)$\wedge$(s=a[0])
Q_4^F = (n>0)$\wedge$(i=1)$\wedge$(s=a[0])$\wedge$(i-n<0)

Given an execution point i, the set of backward program states at i are defined as:

Q_i^B = **pwp**($Stmt_i, Q_{i+1}^B$), if $Stmt_i$ is an assignment statement
Q_i^B = $Stmt_i \wedge Q_{i+1}^B$, if $Stmt_i$ is a branch predicate
Q_{bottom}^B = { }

We illustrate the computation of the set of backward program states for the error trace in Figure 1(b).

Q_1^B = (3-n$\geq$0)$\wedge$(a[0]-a[2=2]$\leq$0)$\wedge$(2-n<0)$\wedge$(a[0]-a[1=1]$\leq$0)$\wedge$(1-n<0)
Q_2^B = (i+2-n$\geq$0)$\wedge$(a[0]-a[i+1=2]$\leq$0)$\wedge$(i+1-n<0)$\wedge$(a[0]-a[i=1]$\leq$0)$\wedge$(i-n<0)
Q_3^B = (i+2-n$\geq$0)$\wedge$(s-a[i+1=2]$\leq$0)$\wedge$(i+1-n<0)$\wedge$(s-a[i=1]$\leq$0)$\wedge$(i-n<0)
Q_4^B = (i+2-n$\geq$0)$\wedge$(s-a[i+1=2]$\leq$0)$\wedge$(i+1-n<0)$\wedge$(s-a[i=1]$\leq$0
Q_5^B = (i+2-n$\geq$0)$\wedge$(s-a[i+1=2]$\leq$0)$\wedge$(i+1-n<0)
Q_6^B = (i+1-n$\geq$0)$\wedge$(s-a[i=2]$\leq$0)$\wedge$(i-n<0)
Q_7^B = (i+1-n$\geq$0)$\wedge$(s-a[i=2]$\leq$0)
Q_8^B = (i+1-n$\geq$0)
Q_9^B = (i-n$\geq$0)
Q_{bottom}^B = { }

Finally, we define the set of actual program states Q_i as $Q_i = Q_i^F \wedge Q_i^B$.

2.5 Detection of Evidence

A predicate A is less restrictive than predicate B if there is some state in A, which is not contained in B, or in other words, $A(x) \Rightarrow B(x)$ is False. An *evidence* at an execution point i indicates that the predicate Q_i representing the set of actual program states is less restrictive than the predicate R_i representing the set of hypothesized program states. We define two types of evidences *explicit* and *implicit*.

Explicit Evidence. An explicit evidence shows that the set of actual program states represented by Q_i and the set of hypothesized program states represented by R_i are disjoint and thus $Q_i \Rightarrow R_i$ is False; or in other words, Q_i is *n*ot stronger than R_i at this program point. Currently, we consider two special cases to detect that the set of states in Q_i and R_i are disjoint. We refer to them as explicit evidence of Type I and explicit evidence of Type II.

Definition: If at an execution point i along a trace, a negative atomic predicate of the form $r : 0\ relop\ const$, i.e., without any variables, appears in the predicate R_i representing the set of hypothesized states, then r constitutes an **explicit evidence** $E_{explicit}(any,$ **r, i) of Type I**.
Let r be a formalized negative predicate in R_i that has the form $(0\ relop\ const)$, i.e., there is no variable involved in the predicate r. For example, `(0 > 2)` is such a *False* predicate. Since Q_i evaluates to *True* and *False* is the strongest predicate, it is obvious that Q_i is less restrictive than r.

Definition: At an execution point i in a trace, let $q : expr_q\ relop\ const_1$ be an atomic predicate in Q_i and $r : expr_r\ relop\ const_2$ be a negative atomic predicate in R_i. Then, q and r form an **explicit evidence** $E_{explicit}(q, r, i)$ **of Type II** iff $expr_q$=$expr_r$.

Since q evaluates to true and r evaluates to False in the given trace, if $expr_q$ = $expr_r$, then for this trace q exercises a state not contained in the set of states represented by r. The symbolic difference between q and r provides us a clue to what modification should be done to the program so as to remove this evidence of $q \Rightarrow r$ being False. For example, for the error trace in Figure 1(b), predicate Q_7^B representing the set of actual program states contains a predicate `(s-a[i=2]≤ 0)` and the predicate R_7 representing the set of hypothesized program states contains another predicate `(s-a[2]≥ 0)`. These two predicates form an explicit evidence $E_{explicit}$`((s-a[i=2]≤ 0), (s-a[2]≥ 0), 7)`. Note that `a[i=2]` and `a[2]` refer to the same variable.

Implicit Evidence. An implicit evidence $E_{implicit}(r)$ is indicated by a negative predicate r in R_1 that is not present in an explicit evidence. For each implicit evidence $E_{implicit}(r)$ in R_1, we consider that the trace is lacking a constraint on r. For example, let consider the R_1 for the postcondition conjunction:

$$R= ((s - a[0] \geq 0)^T \wedge (s - a[1] \geq 0)^F \wedge (s - a[2] \geq 0)^F \wedge (s - a[0] = 0)^T).$$

The corresponding R_1 = $((a[0] - a[1] \geq 0)^F \wedge (a[0] - a[2] \geq 0)^F)$. And,

$$Q_1 = \text{(3-n}\geq\text{0)}\wedge\text{(a[0]-a[2=2]}\leq\text{0)}\wedge\text{(2-n<0)}\wedge\text{(a[0]-a[1=1]}\leq\text{0)}\wedge\text{(1-n<0)}\wedge\text{(n>0)}.$$

However, in this example both the negative predicates in R_1 have corresponding predicates in Q_1 that form explicit evidence of Type II. Therefore, there is no implicit evidence at the top of the trace in this example.

2.6 Location of a Likely Erroneous Statement and Generation of Modification

After an evidence of the predicate Q_i being less restrictive than R_i is detected at an execution point i, the goal is to locate a statement instance at some point j in the trace such that a modification to $Stmt_j$ will remove the evidence at i. From the predicates involved in an evidence, we determine a *problem predicate* and a *correcting predicate*. These two atomic predicates are treated as character strings and the symbolic differences between these two strings are computed. We then use these differences to generate a modification to a statement along the trace so that the detected evidence is removed from the trace. The modified function is tested with the given test suite to check if the

```
Input: An error trace $T$, postcondition conjunction $R$ and test suite $T_s$
Output: $(succ/fail, mod)$, where mod is a modified statement in the program.
procedure $AutoDebug$ $(T, R, T_s)$
        for each execution position $i \in T$ from $i = n$ to 1 do
step1:    Compute set of actual program states $Q_i^*$ and set of hypothesized program states $R_i^*$.
          for each negative predicate $r \in R_i^*$ do
step2:        if an $E_{explicit}(any, r, i)$ detected then   Type I Explicit Evidence
                  Generate and test modifications that change the form of $r$ at $i$.
                  if testing successful then $return(succ, mod)$ endif
step3:        elseif an $E_{explicit}(q, r, i)$ detected then   Type II Explicit Evidence
                  Generate and test modifications that either change the form of $q$ at $i$
                      or change the form of $r$ at $i$ to remove the evidence.
                  if testing successful then $return(succ, mod)$ endif
              endif
          endfor
        endfor
step4:  for each negative predicate $r \in R_1^*$ not present in any explicit evidence do
            Consider Implicit Evidence $E_{implicit}(r)$
            if $E_{implicit}(r)$ indicates a missing loop iteration(s) then
                Generate modification to add missing loop iteration to actual program state.
                Test Modified Program.
                if testing successful then $return(succ = 1, mod)$ endif
            else
                Generate and test modifications that either change form of $r$ at the
                    top of the trace or change the form of a predicate $q$ in $Q_1^{B*}$.
                if testing successful then $return(succ = 1, mod)$ endif
            endif
        endfor
        return $fail$
endprocedure
```

Fig. 2. The AutoDebug algorithm.

error is removed. Otherwise, if possible another modification to remove the evidence is generated. If the evidence cannot be removed by all attempted modifications at the execution point i, the algorithm moves on to process next statement in the trace. Note that our algorithm will terminate successfully if no error trace is generated when the modified function is executed with the given test suite. However, that does not necessarily mean that the modified program is correct. All it means is that the original function was not able to pass all the test cases in the given test suite whereas the modified function is able to pass all the test cases in the test suite. The correctness of the solution is clearly dependent upon how thoroughly the test suite tests the program. It may also be helpful to take input about whether the modification generated by our algorithm will be acceptable to the developer.

3 Description of the Algorithm

In this section, we discuss the steps of our algorithm shown in Figure 2 for automatically locating and correcting an erroneous statement in a function.

Step 1: Compute the Predicates Representing the Sets of Actual and Hypothesized Program States. At the entry of each instance of an executed statement $Stmt_i$ in the given error trace T, we compute the predicate Q_i representing the set of actual program states and the predicate R_i representing the set of hypothesized program states. We apply two rules of inference: *transitivity* and *equality* to deduce new predicates from other predicates in each of the program states. Deduced predicates are added to respective set of program states until no new atomic predicates can be deduced. In the remaining paper, we use Q_i^* and R_i^* to represent the extended sets of Q_i and R_i respectively after including deduced predicates.

Step 2: Detect and Fix Explicit Evidence of Type I. In this type of evidence, there is a *negative* predicate r that does not contain any variables and is present in the predicate R_i^* representing the set of hypothesized program states. For example, let a predicate is `(10=0)` be present in R_i^* at an execution point i on an error trace. Then, it forms an explicit evidence $E_{explicit}(any$, (10=0), i) of Type I.

Generate Modification. The next step is to generate modifications for the statements that would remove the above evidence by changing the form of r at execution point i where evidence is detected. We change the form of r by matching r to a predicate which is implied by the actual program state so that the atomic predicate in the postcondition R corresponding to r will be satisfied if the same trace is followed. We consider the following two approaches to change the form of r at i.

First, if the relational operator in r is =, then we match r to the positive predicate $True$. For relational operator =, we define the $True$ predicate to `0=0`. Note that the form of r can be changed only by an assignment statement between execution point i and the end of trace. It is obvious that modifying an assignment statement in the pwpSlice of r can change the form of r. However, modifying the LHS of an assignment not in the pwpSlice of r can also change the form of r at execution point i. Therefore, we consider each assignment statement between the point i and the $bottom$ of the trace as a possible candidate for modification. Let $Stmt_k$ be the next assignment statement to be considered and let the predicate in R_k^* corresponding to r be r_k. The goal of transforming r to `0=0` can be attained by making $r_k = (0 = 0)$, i.e., **pwp**($Stmt_k, r_{k+1}$)=$(0 = 0)$. It is obvious that if $expr = lhs - rhs$, then **pwp**($lhs = rhs, (expr = 0)$) will be $(0 = 0)$. Therefore, we consider r_{k+1} as the correcting predicate c and the equivalence derived from $Stmt_k$ as the problem predicate e.

We consider e and c as a set of strings of characters and compare them to compute the difference d_e between e and c and the difference d_c between c and e. In our current work, we assume that the error is either on LHS or on RHS of an assignment statement but not on both sides of the assignment statement. If d_e appears on RHS of the assignment statement $Stmt_k$, the modification is generated to replace d_e in $Stmt_k$ by d_c. If d_e appears on LHS of $Stmt_k$, the modification is to replace d_e in $Stmt_k$ by d_c only if d_c is a single variable.

If the evidence cannot be removed by the above modifications, or if the relational operator in r is not =, we then try our second approach to generate modifications explained below. We generate additional modifications by matching r to each predicate q in the predicate Q_i^* with same relational operator as that of r. Note that the predicates in Q_i^* are all positive predicates in the trace, so they are consistent with each other. By matching

r to a predicate in Q_i^* other than q, r becomes consistent with q. Thus, in this case r is the problem predicate e and a predicate q in Q_i^* is used as a correcting predicate c. As before, we compute d_e and d_c. If modifications generated at execution point i are not successful in removing the evidence, as before we propagate this matching to execution points $k > i$ so that the effect of matching at the execution point k is to remove the above evidence at the execution point i. However, there is a difference. Now a matching at k can be performed only if the predicate q_k corresponding to q is present in Q_k^* i.e., it is not killed by some assignment between execution points i and k. In addition, in order to make sure that the modification to an assignment $Stmt_k$ at execution point k will change the form of r at execution point i, we need to check for the following. If the modification is for RHS of an assignment statement $Stmt_k$, then $Stmt_k$ must belong to the pwpSlice of r. However, if the modification is for the LHS of the assignment statement, we need to make sure that after modification, the assignment will appear in the pwpSlice of r.

Test Modification. Each of the modification generated above is applied to the original program. The modified program is then executed for all the test cases in the given test suite. If the modified program passes all the test cases in the test suite then we consider that the error has been corrected. Each of the above modification is tested until a version of the program passes the test suite. If all the above modifications have been tried and the fault is not fixed, the algorithm moves onto to detect next evidence.

Step 3: Detect and Fix Explicit Evidence of Type II. In this step we detect and fix an explicit evidence, in which a predicate q in Q_i^{B*} and a negative predicate r in R_i^* have the same expression on LHS. This evidence also shows that the set of states in Q_i^* are disjoint from the set of states in R_i^*. To illustrate this, let us assume that Q_i^{F*}, Q_i^{B*} and R_i^* at an execution point i on an error trace are given as below.

Q_i^{F*}:(n>0)$\wedge$(i=0)$\wedge$(s=0)$\wedge$(i-n<0)
Q_i^{B*}:(i-n+1$\geq$0)$\wedge$(s-a[i=0]$\leq$0)
R_i^*:(s-a[0]$\geq$0)$^F \wedge$ (s-a[0]=0)^F

Two explicit evidences of Type II are detected at execution point i. They are

$E_1 = E_{explicit}$((s-a[i=0]$\leq$0), (s-a[0]$\geq$0), i)
$E_2 = E_{explicit}$((s-a[i=0]$\leq$0), (s-a[0]=0), i)

For an evidence $E_{explicit}(q, r, i)$ of Type II, either q or r could be in error. Therefore, the modifications for changing the form of q to r at i or changing form of r to q at i are generated. The modifications for changing the form of r are generated in the same manner as described for explicit evidence of Type I. To change the form of q to match to r, we can either change the original branch predicate from which q may be derived, or we can change an assignment statement on the trace. Note that a modification to an assignment statement cannot change the relational operator of q. Therefore, if the relational operators of q and r are different, we directly modify the branch predicate from which r may be derived.

Step 4: Detect and Fix Implicit Evidence. Implicit evidences are detected at the top of the error trace. For each negative atomic predicate r in R_1^* that is not present in any explicit evidence, we form an implicit evidence as $E_{implicit}(r)$. Having an implicit evidence $E_{implicit}(r)$, we check whether the cause for the evidence is because some loop iterations are missing from the trace. If there is a loop in the trace, which contributes some constraints on R_1^*, and the missed constraints have similarity with the constraints added by the loop, then our algorithm attempts to derive the possible missing iterations in the loop and generates modification to a statement that would add those iterations into the trace. This modification is then verified by executing the modified program with the test suite.

If the implicit evidence is not the case of a missing loop iteration(s), the algorithm attempts to remove this evidence from R_1 fix the fault by generating modifications as in Steps 2 and 3. Given an implicit evidence of $E_{implicit}(r)$, modifications to the statements along the trace are generated by matching the negative predicate r to atomic predicates q in Q_1^* and vice versa. As in steps 2 and 3, modifications to the assignment statements in the pwp slice of r are also generated by matching them to the component corresponding to r in the hypothesized state at their exit. As before, each modification is tested by executing the modified program.

4 Experiments

We have implemented our technique using C and Python languages. The autodebug algorithm was implemented in C. Postconditions, preconditions and predicate deduction was implemented using Python. The faulty program is expected to be written in a subset of C using real, integer and character variables, arrays, conditionals and loop control constructs. At present, we do not handle faulty programs using pointers. To handle function calls, we assume that the postconditions and preconditions of the called function are given. We also assume that either the called function does not have errors, or the trace of statements through the called function is available. The faulty program is instrumented to generate execution traces in the format described in the paper. We used the following five programs in our experiments.

Sum: It computes the sum of all integers in an array `a[]`. This problem has simple control structures. The postcondition of this program is a single universal quantifier which is expanded as conjunctions during the execution.

Max: This program (in section 2) searches the maximum element in an unsorted array of integers.

Binary Search: It does binary search on a sorted integer array. Its source code, including the preconditions and postconditions was taken from [7].

Array Copy: This example is a simple program to copy the contents of an array to another array.

Quicksort: This program, taken from [2], is for Quicksort algorithm on an integer array. The original code does not have preconditions and postcondition so we derived them ourselves.

We introduced an error in a statement at a time into these programs. The types of errors introduced include wrong relational operator used in a branch predicate, wrong variable used in a branch predicate, wrong variable used in an assignment statement, incorrect constant used in an assignment statement, etc. We conducted experiments with our algorithm for locating and correcting the erroneous statements. We also experimented with computing program dices [1] for these faulty programs. We tried to limit the modification only to the statements in the computed dice. If the algorithm is not able to correct the program by modification of statements in the dice, then we ran the algorithm without using dice. The heuristic used in [1] for picking a dice is to first form all possible dices and then randomly choose one of them. Since there is no conclusion about which heuristic is better, we used more conservative method. We chose the dice with the largest number of statements so that it most likely will not miss an erroneous statement.

4.1 Results

We show the results of our experiments in Table 1. The column labeled Line No. shows the line number of the statement in the function in which the error was introduced. The column labeled Orig. Stmt. shows the original statement in the correct program. The column labeled Faulty Stmt. shows the statement after error was introduced. The column labeled Fault Type shows the type of fault introduced such as wrong constant, wrong operator, missing variable etc. The last column shows the output obtained from our implementation of AutoDebug algorithm.

It is interesting to note that in rows Sum/1 and Max/3 in Table 1, the correction generated by our algorithm was in a different statement than the one in which error was introduced. However, modification in a statement different from the one in which error was introduced also corrected the problem. Some of the errors resulted in non-terminating loop and they were not corrected by our algorithm. Also, if an error resulted in a loop executing more iterations than required, our algorithm was not able to fix it. Other than these, our algorithm was able to fix most of the errors. Although, we had expected dices to significantly improve the efficiency of the technique, in our experiments we did not find dices to be useful in many cases. Only for 3 errors among all the errors in Table 1, were the erroneous statements included in largest dice for the faulty program. We also tried to consider the union of statements in all dices. However, we did not find dices to be very useful in our examples. The reason was that in many cases, such as in branch predicate fault and variable initialization fault, the faulty statement was executed by all failed trace as well all correct traces. In some cases, no correct traces were generated and hence dices were not helpful. The programs used in our experiments were small and there may be benefits of incorporating dices in our technique for large programs.

5 Related Work

The program slicing based approaches [1,11,15] use *static* or *dynamic dependency analysis* to extract a subset of program statements that can effect the values of variables at the point where a fault is manifested in the program. A novel approach to automatically isolate cause-effect chains, based on the *difference* between the *program states* of a run corresponding to a failed and a successful run, has been recently developed [19,9]. The cause-effect chains isolated by this approach have higher precision than static or dynamic slices. Approaches based on dynamic invariant detection that give programmers warnings that there are anomalies found in the program [8,16] have also been developed. All these approaches assist the programmers by narrowing down the search for erroneous statements to a subset of program statements. However, they do not generate the exact modifications to be made to the program to automatically correct the errors.

Table 1. Results for *Sum, Max,* Binary Search(Bin), Array Copy(Arr) and Quicksort(QS) functions.

Program/	Line	Orig. Stmt.	Faulty Stmt.	Fault Type	Output
Error No.	No.				(Line No., Stmt.)
Sum/1	1	i=0	i=1	const:wrong	(2, s=a[0])
Sum/2	1	i=0	i=2	const:wrong	(1, i=0)
Sum/3	2	s = 0	s = 1	const:wrong	(2, s=0)
Sum/4	4	s=s+a[i]	s = a[i]	var(s):missing	(4, s=s+a[i])
Sum/5	4	s=s+a[i]	s = i+a[i]	var(s):wrong	(4, s=s+a[i])
Sum/6	4	s=s+a[i]	s=s+a[0]	var(a):wrong	(4, s=s+a[i])
Sum/7	4	s=s+a[i]	s=s-a[i]	op:wrong	(4, s=s+a[i])
Sum/8	5	i=i+1	i=i+2	const:wrong	(5, i=i+1)
Sum/9	5	i=i+1	i=i	const:wrong	No (infinite loop)
Sum/10	3	while(i<n)	while(i<n-1)	const:wrong	(3, while(i<n))
Sum/11	3	while(i<n)	while(i<n+1)	const:wrong	No (extra loop)
Sum/12	3	while(i<n)	while(i+n)	relop:wrong	No (infinite loop)
Sum/13	3	while(i<n)	while(i>n)	relop:wrong	No (loop not enter)
Max/1	2	s=0	s=10	const:wrong	(2, s=a[0])
Max/2	5	s=a[i]	s=i	var(a):wrong	(5, s=a[i])
Max/3	1	i=0	i=1	const:wrong	(2, s=a[0])
Max/4	6	i=i+1	i=i-1	op:wrong	No (system error)
Max/5	4	if(s<a[i])	if(s>a[i])	relop:wrong	(4, if(s<a[i])
Max/6	4	if(s<a[i])	if(s<a[0])	var(a):wrong	(4, if(s<a[i]))
Max/7	4	if(s<a[i])	if(s>=a[i])	relop:wrong	(4, if(s<a[i]))
Max/8	3	while(i<n)	while(i<n-1)	branch:const	(3, while(i<n))
Bin/1	1	i=0	i=1	const:wrong	(2, i=0)
Bin/2	2	j=n+1	j=n	const:wrong	(2, j=n+1)
Bin/3	4	e=(i+j)/2	e=i+j	op:wrong	No (infinite loop)
Bin/4	6	i = e	i = j	var(s):wrong	No (infinite loop)
Bin/5	6	i = e	j = e	def(s):wrong	(6, i = e)
Bin/6	4	e=(i+j)/2	e=(i*j)/2	op:wrong	No (infinite loop)
Bin/7	3	while(i+1!=j)	while(i+2<j)	const:wrong	(3, while(i+2<j+1))
Bin/8	3	while(i+1!=j)	while(i!=j)	relop:wrong	No (infinite loop)
Bin/9	5	if(a[e]<=x)	if(a[e]<x)	relop:wrong	(5, if(a[e]<=x))
Bin/10	5	if(a[e]<=x)	if(a[e]>x)	relop:wrong	(5, if(a[e]<=x))
Bin/11	5	if(a[e]<=x)	if(a[0]<=x)	var(a):wrong	(5, if(a[e]<=x))
Bin/12	5	if(a[e]<=x)	if(a[i]<=x)	var(a):wrong	(5, if(a[e]<=x))
Bin/13	5	if(a[e]<=x)	if(i<x)	var(a):wrong	(5, if(a[e]<=x))
Arr/1	3	s1[i]=s2[i]	s1[i]=s2[i+1]	var(a):wrong	(3, s1[1]=s2[i])
Arr/2	3	s1[i]=s2[i]	s1[0]=s2[i]	var(a):wrong	(3, s1[i]=s2[i])
Arr/3	3	s1[i]=s2[i]	s1[i]=i	var(a):wrong	(3, s1[i]=s2[i])
Arr/4	1	i=0	i=1	assign:const	(1, i=0)
Arr/5	4	i=i+1	i=i-1	assign:arithm	No (system error)
Arr/6	2	while(i<=n)	while(i<n)	branch:relop	(2, while(i<n+1))
Arr/7	2	while(i<n)	while(i>n)	branch:relop	No (loop not entered)
QS/1	3	last=(left+right)/2	last=(left+right)*2	assign:arithm	No (out of array boundary)
QS/2	4	temp=a[left]	temp=a[0]	var(a):wrong	(4, temp=a[left])
QS/3	5	a[left]=a[last]	a[left]=temp	var(a):wrong	(5, a[left]=a[last])
QS/4	5	a[left]=a[last]	a[last]=a[last]	var(a):wrong	(5, a[left]=a[last])
QS/5	8	if(a[i]<a[left])	if(a[i]>=a[left])	relop:wrong	(8, if(a[i]<a[left]))
QS/6	10	if(a[i]<a[left])	if(a[i]<a[last])	var(a):wrong	(10, if(a[i]<a[left]))
QS/7	11	a[last]=a[i]	a[left]=a[i]	var(a):wrong	(11, a[last]=a[i])
QS/8	13	a[last]=a[i]	a[left]=a[i]	var(a):wrong	(13, a[last]=a[i])
QS/9	16	i=i+1	i=i	assign:arithm	No (infinite loop)
QS/10	18	temp=a[left]	temp=a[last]	var(a):wrong	(18, temp=a[left])

To determine the exact nature of the error and check whether it lies in the localized program statements, the programmers have to modify the program and re-execute the program until they obtain correct output. In contrast, our approach attempts to automatically locate the error statement and generate the correction to be applied to the erroneous statement. In our future work, we would further analyze the type of errors that can be detected by our approach and the types of errors in which other approaches can be more helpful.

6 Conclusions

In this paper, we have presented a new technique that combines ideas from formal analysis of programs and software testing to automatically locate and correct erroneous statements. Our technique is based on matching of character strings which is guided by removal of some symbolic evidences that make actual program state less restrictive than hypothesized program state at some execution point. Our preliminary experiments show that our approach is promising. In the current work, we have assumed that only one program statement is in error. In our future work, we plan to relax this restriction and evaluate our technique for large programs.

References

1. H. Agrawal, J. R. Horgan, S. London and W. E. Wong, “Fault localization using execution slices and dataflow tests”, *Proceedings of the Sixth IEEE International Symposium on Software Reliability Engineering*, pages 143-151, Toulouse, France, October 1995.
2. R. S. Boyer and J. S. Moore “A Computational Logic Handbook”, *Academic Press*, Boston.
3. L. A. Clarke and D. J. Richardson, “The application of error-sensitive testing strategies to debugging”, *Proceedings of the Symposium on High-Level Debugging*, pages 45-52, 1983.
4. R. A. DeMillo, H. Pan and E. H. Spafford, “Critical slicing for software fault localization”, *Proceedings of the International Symposium on Software Testing and Analysis*, pages 121–134, San Diego, CA, 1996.
5. E. W. Dijkstra, “A Discipline of Programming”, *Prentice Hall*, Englewood Cliffs, NJ, 1976.
6. C. Ghezzi, M. Jazayeri and D. Mandrioli, “Fundamentals of Software Engineering”, Second Edition, *Prentice Hall*, 2003.
7. D. Gries, “The Science of Programming”, *Springer-Verlag*, 1981.
8. S. Hangal and M. S. Lam, “Tracking down software bugs using automatic anomaly detection”, *Proceedings of the International Conference on Software Engineering*, May, 2002,
9. R. Hildebrandt and A. Zeller, “Simplifying failure-inducing input”, *Proceedings of the International Symposium on Software Testing and Analysis*, pages 135-145, 2000.
10. J. A. Jones, M. J. Harrold and J. Stasko, “Visualization of test information to assist fault localization”, *Proceedings of the 24th International Conference on Software Engineering*, Orlando, Florida, USA, May 2002, pp. 467-477.
11. B. Korel and J. Rilling, “Application of dynamic slicing in program debugging”, *Automated and Algorithmic Debugging*, pages 43-58, 1997.
12. K. R. M. Leino, J. B. Saxe and R. Stata, “Checking Java programs via guarded commands”, *Compaq SRC Technical Note # 1999-002*, Palo Alto, CA, 1999.
13. K. R. M. Leino, T. Millstein, and J. B. Saxe. “Generating error traces from verification-condition counterexamples”. *http://research.microsoft.com/ leino/papers.html*.
14. R. Lencevicius, “On-the-fly query-based debugging with examples”, *Automated and Algorithmic Debugging*, 2000.
15. J. R. Lyle and M. Weiser, “Automatic program bug location by program slicing”, *Proceedings of the 2nd International Conference on Computers and Applications*, pages 877-882, 1987.
16. J. W. Nimmer and M. D. Ernst. “Invariant inference for static checking”, *Proceedings of the ACM/SIGSOFT Symposium on the Foundations of Software Engineering*, pages 11-20, Charleston, SC, November 2002.
17. M. Renieris and S. Reiss, “Fault localization with nearest neighbor queries”, *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, October 2003.
18. E. Y. Shapiro, “Algorithmic Program Debugging”, *The MIT Press*, 1983.
19. A. Zeller, “Isolating cause-effect chains from computer programs”, *Proceedings of the ACM/SIGSOFT International Symposium on Foundations of Software Engineering*, 2002.

Filtering TOBIAS Combinatorial Test Suites

Yves Ledru, Lydie du Bousquet, Olivier Maury, and Pierre Bontron

Laboratoire Logiciels, Systèmes, Réseaux - IMAG
B.P. 72 - F-38402 - Saint Martin d'Hères Cedex - France
{Yves.Ledru,lydie.du-bousquet}@imag.fr

Abstract. TOBIAS is a combinatorial testing tool, aimed at the production of large test suites. In this paper, TOBIAS is applied to conformance tests for model-based specifications (expressed with assertions, pre and post-conditions) and associated implementations. The tool takes advantage of the executable character of VDM or JML assertions which provide an oracle for the testing process. Executing large test suites may require a lot of time. This paper shows how assertions can be exploited at generation time to filter the set of test cases, and at execution time to detect inconclusive test cases.

Keywords: combinatorial testing, model-based specifications, VDM, JML

1 Introduction

Software testing appears nowadays as one of the major techniques to evaluate the conformance between a specification and some implementation. Some may argue that testing only reveals the presence of errors and that conformance may only be totally guaranteed by formal proof, exhaustive testing or a combination of both techniques. Unfortunately, such techniques are often very difficult to apply. In such cases, testing may contribute to increase the confidence that the implementation conforms to its specification. Confidence may result from coverage measurements, from the principles of the test synthesis or selection technique, from the size of the test suite, or from the expertise of the test engineers.

Industrial experiments [5] have shown that test cases within a large test suite often feature a high level of similarity. Many test cases correspond to the same sequence of method calls, with different parameters. Producing these test cases is a repetitive task that reveals the need for appropriate tool support.

From these observations, we have developed the TOBIAS test generator[1] which is aimed at the production of a large set of similar test cases. TOBIAS starts from a test pattern and a description of its instantiations. The tool then unfolds the pattern into a large set of test cases which can be output according to the format of several test tools: calls to VDM operations [12] for VDMTools, Java test cases for JUnit[9] and JML specifications [8,10,11], and test purposes for TGV [7].

[1] TOBIAS was developed within the COTE project, with the support of the French RNTL program. The COTE project gathered Softeam, France Telecom R&D, Gemplus, IRISA and LSR/IMAG.

TOBIAS is a typical example of combinatorial testing tool. Its originality is to deal with sequences of method calls, instead of only combination of parameter values. This allows to use the tool with systems that require several interactions before reaching some "interesting" states. It also allows to design test cases in terms of the behavior that has to be exercised.

This paper gives an introduction to TOBIAS. Sect. 2 recalls the principles of conformance testing using executable model-based specifications. Then Sect. 3 gives a quick presentation of the tool and reports on its capability to find errors, on the basis of a simple example, and from the results of industrial experiments. The intrinsic limitation of the tool is that it is subject to combinatorial explosion. Sect. 4 presents two kinds of filters that can be used with TOBIAS to help master the size of test suites. Finally, Sect. 5 draws the conclusions and perspectives of this work.

2 Conformance Testing with Model-Based Specifications

2.1 Checking Conformance with Model-Based Specifications

Model-based specifications describe a system in terms of invariant properties, pre- and postconditions. Some model-based languages, e.g. VDM and JML, have an executable character. It is thus possible to use invariant assertions, as well as pre- and postconditions as oracle for a conformance testing process. VDM assertions can be evaluated in the VDMTools environment against the VDM version of the specified code [6], or compiled into C++ [1]. JML specifications are translated into Java, added to the code of the specified program, and checked against it. The executable assertions are thus executed before, during and after the execution of a given operation (Fig. 1).

One should note that the specification invariants are not exactly checked at the same instants in JML and VDM. In VDM, invariants are evaluated after each statement. In JML, invariants are properties that have to hold in all *visible states*. A visible state roughly corresponds to the initial and final states of any method invocation [8].

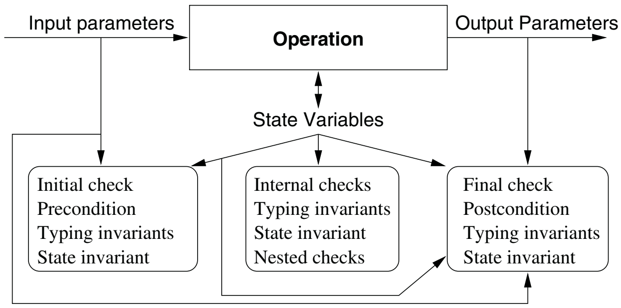


Fig. 1. Dynamic checks associated to operation execution

When an operation is executed in one of those environments, three cases may happen (Fig. 1):

– All checks succeed: the behavior of the operation conforms with the specification for these input values and initial state. The test delivers a PASS verdict.
– An intermediate or final check fails: this reveals an inconsistency between the behavior of the operation and its specification. The implementation does not conform to the specification and the test delivers a FAIL verdict.
– An initial check fails: in this case, performing the whole test will not bring useful information because it is performed outside the specified behavior. This test delivers an INCONCLUSIVE verdict.
  For example, $\sqrt{x}$ has a precondition that x has to be positive. Therefore, a test of a square root method with -1 leads to an INCONCLUSIVE verdict.

2.2 A Small Example in VDM and JML

Let us study a simple example of buffer system (Fig. 2). This system is composed of three buffers. The specification models only the number of elements present in the buffers. A buffer is then modeled with an integer value, which indicates the number of elements in it. The system state is given by the three variables `b1`, `b2` and `b3`.

The maximum size of the system is 40 elements. The system has to distribute the elements amongst the buffers so that: buffer `b1` is smaller than `b2`, which is smaller than `b3`. The difference between `b1` and `b3` should not exceed 15 elements. These constraints leave some freedom on the way to share the elements between buffers. For example, 30 elements can be stored as `b1=5 b2=10 b3=15` or as `b1=8 b2=10 b3=12`.

Three methods are set to modify the systems:

– `Init` resets all buffers to zero.
– `Add(x)` increases the total number of elements of the system of a strictly positive number (`x`) (i.e. it adds `x` elements to the buffers; these elements are distributed in `b1`, `b2`, and `b3`).
– `Remove(x)` decreases the total number of elements in the system of a strictly positive number (`x`) (i.e. it removes `x` elements from the buffers).

The specifications of `Add` and `Remove` keep some implementation freedom: the buffer in which the elements have to be added/removed is not set. For example, if the current state is `8 10 12`, and if 2 elements have to be added, the final state could be `8 10 14`, `8 12 12` , but also `6 12 14`.

2.3 Test Cases

We define a test case as a sequence of operation calls. For example, the following test case initializes the buffer system, adds two elements and removes one of them.

```
TC1 : Init() ; Add(2) ; Remove(1)
```

```
------------------ VDM ------------------
state buffers of
        b1 : nat
        b2 : nat
        b3 : nat

inv mk_buffers(b1,b2,b3) ==
        b1+b2+b3<=40 and 0<=b1 and b1<=b2 and b2<=b3 and b3-b1<=15
init B == B = mk_buffers(0,0,0)
end

operations
Init:() ==> ()
Init() == ...
post b1+b2+b3=0
;
Add: nat ==> ()
Add(x) == ...
pre x<=5 and b1+b2+b3+x<=40
post b1+b2+b3 = b1~+b2~+b3~+x
;
Remove: nat ==> ()
Remove(x) == ...
pre x<=5 and x<=b1+b2+b3
post b1+b2+b3 = b1~+b2~+b3~-x
;
------------------ JML ------------------
public class Buffer{
    public int b1;
    public int b2;
    public int b3;

    /*@ public invariant
      @ b1+b2+b3<=40 && 0<=b1 && b1<=b2 && b2<=b3 && b3-b1<=15; */

    /*@ requires true;
      @ modifies b1, b2, b3;
      @ ensures b1==0 && b2==0 && b3==0; */
    public Buffer(){}

    /*@ requires true;
      @ modifies b1, b2, b3;
      @ ensures b1==0 && b2==0 && b3==0;   */
    public void Init(){...}

    /*@ requires x<=5 && b1+b2+b3+x<=40 && x>=0;
      @ modifies b1, b2, b3;
      @ ensures b1+b2+b3==\old(b1+b2+b3)+x;
    */
    public void Add(int x){...}

    /*@ requires x<=5 && x<=b1+b2+b3 && x>=0;
      @ modifies b1, b2, b3;
      @ ensures b1+b2+b3==\old(b1+b2+b3)-x;  */
    public void Remove(int x){...}
}
```

Fig. 2. Buffer example specification in VDM and JML

Each operation call may lead to a PASS, FAIL or INCONCLUSIVE verdict. As soon as a FAIL or INCONCLUSIVE verdict happens, we choose to stop the test case execution and mark it with this verdict. A test case that is carried out completely receives a PASS verdict.

For example, in the context of the above specification, the test cases TC2 and TC3 should produce an INCONCLUSIVE verdict. If test TC4 is executed against a "correct" implementation, it should produce a PASS.

```
TC2 : Init() ; Add(-1)
TC3 : Init() ; Add(2) ; Remove(3)
TC4 : Init() ; Add(3) ; Remove(2) ; Remove(1)
```

3 TOBIAS

TOBIAS is a test generator based on combinatorial testing [4]. Combinatorial testing performs combinations of selected input parameters values for given operations and given states. For example, a tool like JML-JUnit [3] generates test cases which consist of a single call to a class constructor, followed by a single call to one of the methods. Each test case corresponds to a combination of the parameters of the constructor and a combination of the parameters of the method.

3.1 Principles of TOBIAS

TOBIAS adapts combinatorial testing to the generation of sequences of operation calls. This allows to reach states that do not correspond to a single call to a constructor. It also allows to design tests in terms of behaviors rather than states. For example, in the specification of the buffers, the initial state is fixed (`0 0 0`), and it is not possible to add more than 5 elements at a time. Therefore a rather long sequence is needed (at least 8 operations) to test the behavior of the system at its limits (40 elements).

The input of TOBIAS is composed of a test pattern (also called test schema) which defines a set of test cases. A pattern is a bounded regular expression involving the operations of the VDM or JML specification. TOBIAS unfolds the pattern into a set of sequences, and then computes all combinations of the input parameters for all operations of the pattern.

The patterns may be expressed in terms of *groups*, which are structuring facilities that associate a method, or a set of methods to typical values. For example, let us consider schema `S1`:

$$\begin{cases} \texttt{Init() ; Add\_Gr} \\ \text{with } \texttt{Add\_Gr} = \{\texttt{Add}(x) | x \in \{1,2,3,4,5\}\} \end{cases}$$

`Add_Gr` is a set of 5 instantiations of calls to the method `Add`. The pattern `S1` is unfolded into 5 test sequences:

```
S1-TC1 : Init() ; Add(1)
S1-TC2 : Init() ; Add(2)
...
S1-TC5 : Init() ; Add(5)
```

Groups may also involve several operations. Let S2 and S2' be two other examples of schemas:

$$\begin{cases} \texttt{S2} = \texttt{Init() ; Modify\_Gr\^{}\{1..2\}} \\ \texttt{S2'} = \texttt{Init() ; Add(2) ; Modify\_Gr\^{}\{1..2\}} \\ \text{with } \texttt{Modify\_Gr} = \{\texttt{Add}(x) | x \in \{1,2,3,4,5\}\} \cup \{\texttt{Remove}(y) | y \in \{1,3,5\}\} \end{cases}$$

Modify_Gr is a set of (5+3)=8 instantiations. The expression ^{1..2} means that the group is repeated 1 to 2 times. The patterns S2 and S2' are unfolded into 8+(8*8)=72 test sequences:

```
S2-TC1 : Init() ; Add(1)
  ...
S2-TC8 : Init() ; Remove(5)
S2-TC9 : Init() ; Add(1) ; Add(1)
  ...
S2-TC72 : Init() ; Remove(5) ; Remove(5)
         ----------------------
S2'-TC1 : Init() ; Add(2); Add(1)
  ...
S2'-TC72 : Init() ; Add(2); Remove(5) ; Remove(5)
```

Group definitions may be reused in several schemas, leading to some level of modular construction.

3.2 Finding Errors with Tobias

Let us consider the buffer problem specification. We have proposed an implementation, containing one error: the Remove operation can set one of the three buffers to a negative value while keeping the total number of elements positive, which is forbidden by the specification invariant. This solution was implemented in VDM and Java. We executed the tests corresponding to the schemas S2, S2', S3, and S4.

$$\begin{cases} \texttt{S3} = \texttt{Init() ; Add(5)\^{}7 ; Modify\_Gr\^{}\{1..2\}} \\ \texttt{S4} = \texttt{Init() ; Add\_Gr ; Modify\_Gr\^{}\{1..3\}} \\ \text{with } \texttt{Modify\_Gr} = \{\texttt{Add}(x) | x \in \{1,2,3,4,5\}\} \cup \{\texttt{Remove}(y) | y \in \{1,3,5\}\} \\ \text{and } \texttt{Add\_Gr} = \{\texttt{Add}(x) | x \in \{1,2,3,4,5\}\} \end{cases}$$

The schema S2' was introduced in order to decrease the number of inconclusive verdicts of schema S2. The schema S3 aims at testing the behavior of the application at the "limits", i.e. when the buffer system is quite full. The schema S4 was built to produce lots of test sequences (some kind of "brute force approach"). The following table gives the verdicts of the various test cases.

Schema	Test cases	Pass	Inconclusive	Fail
S2	72	39	33	0
S2'	72	48	22	2
S3	72	57	15	0
S4	2920	1887	773	260

As expected, the error is detected (by schemas `S2'` and `S4`). `S3` is aimed at testing full buffers and can not reveal the error; `S2` is a small test suite with a lot of inconclusive test cases, which does not achieve enough exhaustiveness to find the error.

This example shows that TOBIAS test suites are able to find errors. Here the error was not straightforward, and small test suites such as `S2` are not sufficient to detect it (actually, the error may only happen if two `Add` operations have been performed). Longer test sequences are needed, such as the ones generated by `S4`.

We have carried out several experiments with TOBIAS. In [12], we report on a VDM case study. This case study showed that the development of a TOBIAS test suite requires the same amount of effort as a simple manual test suite. It also shows that since TOBIAS test suites achieve more exhaustiveness (by exercising all combinations in the schema), they reveal some errors that are often overlooked by manual test suites.

3.3 Industrial Case Study

Two experiments were also carried out on an industrial case study provided by Gemplus (a smart card manufacturer). The case study is a banking application which deals with money transfers. It has been produced by Gemplus Research Labs and is somehow representative of java applications connected to smart cards. The application user (i.e. the customer) can consult his accounts and make some money transfers from one account to another. The user can also record some "transfer rules", in order to schedule regular transfers. These transfer rules can be either saving or spending rules.

The case study is actually a simplified version of an application already used in the real world. The code length is 500 lines. The specification was given in JML. Most preconditions are set to true. Since the application deals with money, and since some users may have malicious behaviors, the application is expected to have defensive mechanisms. Thus, it is supposed to accept any entry, but it should return error messages or raise exceptions if the inputs are not those expected for a nominal behavior. It is a typical example of *defensive programming* style. This means that test cases do not produce INCONCLUSIVE verdicts.

Two testing experiments with TOBIAS were carried out from this case study. The first one was carried out by a Gemplus team. They have first used their internal testing methodology to elaborate an informal test plan. It includes 40 nominal "scenarios" (a scenario is an informal description of a test case). It was possible to abstract those scenarios and express them with only 5 TOBIAS schemas, which were unfolded into 1900 executable test cases. This experiment shows that TOBIAS schemas are more compact than test cases. Moreover, by abstracting test cases into schemas, we ended up with more general schemas than the original scenarios, resulting into 50 times more test cases.

This experiment was considered as a success by our industrial partner. On the one hand, TOBIAS schemas were perceived as an interesting structuring mechanism for the design of tests. On the other hand, the tool allows to complete the original test suites by achieving some kind of exhaustiveness.

The second experiment was carried out by our research team. From the informal requirements, we deduced 17 TOBIAS schemas, mainly to simulate malicious behaviors. They were unfolded into 1100 test sequences, representing 40 000 Java code lines (for JUnit). It took 6 person-day to analyze the specification, produce the abstract scenarios, execute the tests and analyze the traces. (The test suite execution time by itself takes only 1 hour.) The execution of the test cases revealed 16 errors, in either the Java code or in the corresponding JML specification. A discussion with the Gemplus team after the experiment showed that we discovered most of the errors in the code. The only remaining error was impossible to detect because the JML specification did not address this feature of the system.

3.4 Conclusion

TOBIAS is a combinatorial tool that instantiates a large set of test sequences from an abstract description. It aims to be a simple and easy to use tool for combinatorial testing which supports and amplifies the creative work of a test engineer. The tool can also be used in order to express existing test sequences in a more abstract way, which helps the test engineer to structure his test suite.

Several experiments have shown that it is well-suited for a conformance testing activity, in conjunction with executable model-based specification. These experiments include both research and industrial case studies. In all these case studies, the tool allowed to detect errors in the implementation under test.

4 Handling Large Test Suites

The major strength of TOBIAS is also its main weakness. The combinatorial approach allows to produce large test suites, whose systematic character helps to detect errors. But the size of the test suite may also become a problem when too many resources are needed to run the tests and analyze their results. The first way to avoid combinatorial explosion is to design test schemas with great care. By avoiding useless calls in the schema and by keeping the possible values of a parameter to a minimum, we were able to control the number of generated test cases in the experiments we led so far. Nevertheless, two additional mechanisms have proved to be useful to reduce the amount of tests. They filter the set of test cases either at execution or at generation time.

The typical size of a TOBIAS test suite ranges from hundreds to thousands of tests. Today, the largest test suite generated by the tool counts about 40 000 test cases. Several experiments have shown that such test suites include a large number of inconclusive test cases. For instance, schema `S4` leads to 773 INCONCLUSIVE test cases. Although these are useful to test preconditions, their execution may require a significant amount of time, and it makes sense to try to eliminate some of them.

This section will discuss two techniques, based on predicates, that are used to control or cope with the size of TOBIAS test suites.

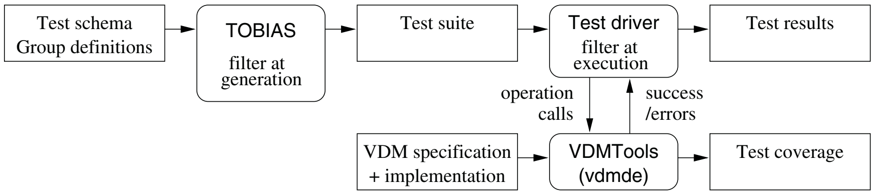


Fig. 3. Exploitation of TOBIAS generated tests in the VDM environment

– Filtering at the execution time: a test driver takes into account the results of the oracle and filters out test cases with a prefix that has already failed one of the checks.
– Filtering at the generation time: the test case generator can take into account a predicate which filters out test cases whose input parameters do not fulfill the predicate.

4.1 Filtering at Execution Time

By construction, TOBIAS test suites are made up of similar test cases. One of the possible similarities is that several test cases may share a common prefix. For example, schema `S2'` includes 9 test cases which start with prefix `init() ; Add(2) ; Remove(5)`. If the execution of the prefix is erroneous (or INCONCLUSIVE) for any of the 9 test cases, and if the implementation is deterministic, the 8 remaining test cases will also exhibit an erroneous (or INCONCLUSIVE) prefix. It is useless to execute the test cases with this prefix.

Therefore, we have developped test drivers that take this property into account. Every erroneous prefix is stored during the execution of the test suite. Before playing a new test case, it is compared to the stored erroneous prefixes and discarded if it matches any of them.

The nice property of this filtering scheme is that it takes advantage of the executable VDM/JML assertions (mainly preconditions) to help filter the test suite. It does not require additional input from the user. Of course, this filtering scheme is better suited to specifications that include strong preconditions. It will not provide any benefit for specifications which adopt a defensive programming style, with all preconditions set to true.

Buffer Case Study

The following table shows the execution time for the 4 test suites of the buffer problem. The tests were executed on a Pentium III/500MHz/128Mb linux machine.

					VDM		JML	
Schema	Test cases	Pass	Inconclusive	Fail	Exec. Time	Exec. Time with filtering	Exec. Time[2]	Exec. Time with filtering
`S2`	72	39	33	0	2 s.	1.205 s.	0.709 s.	0.134 s.
`S2'`	72	48	22	2	2 s.	1.674 s.	0.524 s.	0.157 s.
`S3`	72	57	15	0	6 s.	4.596 s.	0.400 s.	0.269 s.
`S4`	2920	1887	773	260	2min 05 s.	9.930 s.	14.307 s.	1.352 s.

[2] With JML-Junit environment.

The tests were executed with filtering and non-filtering test drivers. As expected, the optimized drivers execute the test suites quicker than the original test drivers. The speed up is more important when there are many INCONCLUSIVE (or FAIL) verdicts. Both kinds of drivers reveal the implementation error.

Banking Application

The banking application is a typical example of defensive programming. The preconditions of operations are usually set to `true`, in order to face all kinds of unexpected inputs. With such applications, the test cases never end up with an INCONCLUSIVE verdict. Therefore, filtering at execution time can only take into account the prefixes which lead to a FAIL verdict. In the banking application, this corresponds to a small number of tests (at most 1%). Hence, filtering at execution time does not lead to a significant speed-up.

The following experiment was led to make sure that the filtering mechanism did not slow down execution significantly when there are no INCONCLUSIVE test cases. We have executed the tests with both JUnit and our driver for Java, on a Pentium III/500MHz/128Mb Windows machine. This one has some limitations. For instance, it is not possible to set several instantiations for a constructor method in the same test suite (a test suite here is the set of test cases derived from one schema). As a result, some the test suites were not executable with our driver. The following table shows the execution time for the tests. As it can be noticed, the execution time with our driver is shorter than with JUnit.

Schema	nb of tests	with Junit	with our driver	Speedup
One account creation	162	0.671 s.	0.410 s.	0.39
Several account creations	96	0.401 s.	0.030 s.	0.93
One account deletion	30	0.160 s.	0.060 s.	0.63
Several account deletions	512	2.553 s.	0.641 s.	0.75
Several transfers	1	0.050 s.	0.060 s.	-0.20
Incorrect transfer (1)	16	0.251 s.	0.240 s.	0.04
Incorrect transfers (2)	60	0.520 s.	0.601 s.	-0.16
Incorrect transfers (3)	2	0.040 s.	0.030 s.	0.25
Use of infinity values	12	0.141 s.	0.110 s.	0.22
12 digit numbers	4	0.070 s.	0.060 s.	0.14
Transfers and account deletion	12	0.140 s.	0.080 s.	0.43
Transfer rules (1)	120	2.163 s.	2.204 s.	-0.02
Transfer rules (2)	96	1.733 s.	1.592 s.	0.08
Transfer rules (3)	12	0.341 s.	0.180 s.	0.47
Transfer rules (4)	8	0.301 s.	0.110 s.	0.63
Saving rule and account deletion	3	0.591 s.	0.050 s.	0.91
Spending rule and account deletion	3	0.711 s.	0.080 s.	0.87

Our experiments (the banking application and the buffers) show that our test driver is faster than JUnit. There are several reasons:

- algorithmic reasons: when a large number of tests have the same prefix, and when this prefix leads to a FAIL or an INCONCLUSIVE verdict, these tests (which are amongs the longest of the test suite) are not executed with our driver. For example, in the `S4` schema, 760 tests are discarded, which corresponds to a quater of the tests.
- technical reasons: JUnit has a generic character and uses introspection/reflection facilities to discover th tests stored in a class. Our test driver is directly compiled from the test suite and does not have to find this information. Moreover, we suspect that the graphical interface of JUnit (which were used during our tests) also slows down the execution. The banking experiment, which never leads to INCONCLUSIVE verdicts, shows that these technical reasons alone result in significant speedups.

4.2 Filtering Test Cases at Generation Time

The previous section has shown that preconditions and other assertions could filter a lot of INCONCLUSIVE test cases at execution time. TOBIAS provides an other mechanism which allows to eliminate some test cases at generation time, using a VDM predicate as a filter.

Let us consider again the schemas `S2` to `S4`. A lot of the inconclusive verdicts are due to the fact that the total number of removed elements is greater than the total number of added elements. One idea is to complete the schema definitions with a constraint on the combination of parameters. Schema `S2` was defined as:

$$\begin{cases} \texttt{S2} = \texttt{Init() ; Modify\_Gr\^{}\{1..2\}} \\ \text{with } \texttt{Modify\_Gr} = \{\texttt{Add}(x) | x \in \{1,2,3,4,5\}\} \cup \{\texttt{Remove}(y) | y \in \{1,3,5\}\} \end{cases}$$

and unfolds into 72 test cases:

```
S2-TC1 : Init() ; Add(1)
...
S2-TC72 : Init() ; Remove(5) ; Remove(5)
```

Let `add1` be the sequence of `x` parameters associated to each call to `Add` in a given test case, and `del1` be the sequence of `y` parameters associated to each call to `Remove`. For example, test case `S2-TC1` corresponds to `add1 = [1]` and `del1 = []`, and test case `S2-TC72` corresponds to `add1 = []` and `del1 = [5,5]`. The following VDM constraint expresses that the sum of the elements of `add1` is greater than or equal to the sum of elements of `del1`.

```
S2_constraint : () ==> bool
S2_constraint() == (
  dcl sommeAdd : nat:=0;
  dcl sommeDel  : nat:=0;
  for a in add1 do (sommeAdd:= sommeAdd+a;);
  for d in del1 do (sommeDel:= sommeDel+d;);
return(sommeAdd>=sommeDel) )
```

TOBIAS has been extended to generate sequences `add1` and `del1` for each unfolded test case, and then pass them to a VDM interpreter which evaluates the constraint. Test cases which fail to verify the constraint are discarded from the generated test suite.

With schema `S2` and `S2_constraint`, the resulting test suite only features 48 test cases, among wich only 9 lead to INCONCLUSIVE verdict (instead of 33).

This first example has shown that constraints can get rid of INCONCLUSIVE tests at generation time. But this technique requires the test engineer to write the constraint, while filtering at execution time took advantage of the existing preconditions. Still, filtering at generation time is an interesting mechanism, because constraints can be motivated by other concerns than simply ruling out INCONCLUSIVE tests, as will be shown in the following example.

Application to the Banking Problem

It was already mentioned that the banking problem does not lead to INCONCLUSIVE verdicts. Still, constraints can be used in this case study to master combinatorial explosion by adding further test hypotheses.

One of the 17 schemas is named “Several account deletions”. It is unfolded into 512 test cases, which is actually the highest number of test cases in this experiment. This schema is defined as follows:

```
Create^{2..2}; Delete^{3..3}; CreateErr; Delete
```

where `Create` has only one instance, `CreateErr` has two instances and `Delete` has four instances corresponding to four possible values of its only integer parameter. This schema is unfolded thus into 1*1*4*4*4*2*4= 512 test cases.

In order to reduce this size, one may wish to express additional test hypotheses. For example, `Delete` can be instantiated as `Del(10)`, `Del(11)`, `Del(12)`, or `Del(13)`. A first test hypothesis may be that the order of the first three instances of `Delete` is not significant. Therefore the following test sequences are equivalent for the tester:

```
Create; Create; Del(10); Del(11); Del(12); CreateErr; Del(10)
Create; Create; Del(12); Del(11); Del(10); CreateErr; Del(10)
```

Let `del1` be the sequence of parameters associated to the first three calls to `Delete`, the following constraint expresses that only the sequence where the parameters appear in ascending order will be kept:

```
forall val1, val2 in set inds del1 &
               val1<val2 => del1(val1)<=del1(val2)
```

Another test hypothesis (here a regularity hypothesis) is that it does not make sense to try to delete the same account more than twice. This hypothesis can be enforced if the four `Delete` calls refer to at least three different accounts. Let `del2` be the single element sequence corresponding to the fourth call to `Delete`, this constraint can be expressed as:

```
card(elems(del1) union elems(del2))>=3
```

These hypotheses are then grouped into the following constraint.

```
Delete_C : () ==> bool
Delete_C () == (
 return(
        (forall val1, val2 in set inds del1 &
              val1<val2 => del1(val1)<=del1(val2))
        and
         card(elems(del1) union elems(del2))>=3
        )
)
```

When TOBIAS takes this constraint into account, the number of unfolded test cases is reduced from 512 to 80. From a test engineer point of view, this reduction may be interesting since it results in a better balance of the whole test suite. Thus this test schema no longer appears as the most significant one.

5 Conclusion

This paper has presented TOBIAS, a test case generator based on the combinatorial unfolding of test schemas. It has shown how the tool can be combined with executable model-based specifications in a conformance testing process. TOBIAS aims to be a simple and easy to use tool for combinatorial testing which supports and amplifies the creative work of a test engineer. The tool has proved to be useful to detect errors in several case studies, including an industrial experiment where Java code was tested against a JML specification.

Other tools adopt a combinatorial testing approach in combination with model-based specifications. Korat [2] and JML-JUnit [3] generate combinations of a call to a constructor followed by a single call to one of the methods of the class. Korat uses an elaborate generator to cover a wide range of calls to the constructor. TOBIAS adds the possibility to express a sequence of method calls in the test schema, allowing to reach states that cannot be created with the constructor and to express tests on the basis of a behavior.

This paper has also presented filters that help master the size of the generated test suites. Filtering at execution time is an interesting feature because it does not require additional inputs from the test engineer. It allows to filter a significant percentage of the tests for specifications with strong preconditions.

Filtering at generation time requires that the test engineers express some constraints on the schema parameters. But it is a more flexible filtering mechanism and allows to translate test hypotheses into filtering constraints.

Perspectives. Several improvements may be considered when filtering at generation time. On the one hand, several typical constraints could be added as primitives of the schema language. For example, a variant of iteration of a method could mandate parameter values to be all different, or to appear in ascending order. On the other hand, a library of constraints could be developed to express frequently used testing constraints. Moreover, since constraints translate test hypotheses, the library could be structured in terms of these higher level concerns.

Still, improvements in filtering capabilities should not prevent the test engineers from handling combinatorial explosion by a careful design of their test schemas. Further methodological advances are needed to guide the elaboration of test schemas. We expect that further experiments with TOBIAS with help us to progress in that direction.

References

1. B.K. Aichernig. Automated black-box testing with abstract VDM oracles. In M. Felici, K. Kanoun, and A. Pasquini, editors, *SAFECOMP'99*, LNCS 1698, pages 250–259. Springer, 1999.
2. C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis*, Rome, 22–24 July 2002. IEEE.
3. Y. Cheon and G.T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In B. Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Malaga, Spain, Proceedings*, LNCS 2474, pages 231–255. Springer, 2002.
4. D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):83–88, 1996.
5. L. du Bousquet, H. Martin, and J.-M. Jézéquel. Conformance Testing from UML specifications, Experience Report. In Gesellschaft für Informatik, editor, *p-UML workshop, Lecture Notes in Informatics*, volume P-7, pages 43–56, Toronto, 2001.
6. The VDM Tool Group. VDM-SL Toolbox User Manual. Technical report, IFAD, October 2000. ftp://ftp.ifad.dk/pub/vdmtools/doc/userman_letter.pdf.
7. T. Jéron and P. Morel. Test Generation Derived from Model-checking. In *Computer Aided Verification (CAV)*, LNCS 1633. Springer, 1999.
8. The Java Modeling Language (JML) Home Page. http://www.cs.iastate.edu/~leavens/JML.html.
9. JUnit. http://www.junit.org.
10. G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.
11. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06q, Iowa State University, Department of Computer Science, June 2002.
12. O. Maury, Y. Ledru, P. Bontron, and L. du Bousquet. Using TOBIAS for the automatic generation of VDM test cases. In *Third VDM Workshop (in conjunction with FME'02)*, july 2002.

Systematic Testing of Software Architectures in the C2 Style

Henry Muccini[1], Marcio Dias[2], and Debra J. Richardson[2]

[1] Universita' degli Studi dell'Aquila
Dipartimento di Informatica
Via Vetoio, 1 – L'Aquila - Italy
muccini@di.univaq.it
[2] School of Information and Computer Science
University of California, Irvine
Irvine, CA, USA
{mdias,djr}@ics.uci.edu

Abstract. The topic of software architecture (SA) based testing has recently raised some interest. Recent work on the topic has used the SA as a reference model for code conformance testing, to check if an implementation fulfills (conforms to) its specification at the SA level. In this context, on previous papers, we have analyzed: i) how suitable test cases can be "selected" from the SA specification and ii) how they may be "refined" into concrete tests executable at the code level. While the selection stage has been done systematically, the refinement step has been left to be done manually, based on the software engineer knowledge on how to map "abstract values of the specification to the concrete values of the implementation". In this paper, we extend previous approaches, by providing a systematic way to perform the refinement step. We show how choosing a specific architectural style, which supports implementation and facilitates the mapping among SA-based and code-based test cases, a completely systematic SA-based testing approach can be delivered.

1 Introduction

Software testing consists of the "dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the specified behavior" [5]. Traditional approaches to software testing select test cases based on the source code of the program to be tested [16].

With the advent and use of software specifications, source code no longer has to be the single source for selecting test cases: formal, informal and model-based specifications can be used for this purpose [5, 16]. The importance of the use of formal methods in software specification and design does not need to be stressed here. Several authors have highlighted the advantages of formal methods in testing, and different techniques have been proposed to select tests from semi-formal specifications [24], algebraic specifications [3], model-based specifications [12] and Software Architecture-based specifications [4, 25].

Particular interest has been devoted to *specification-based conformance testing* [8, 14, 30]. Conformance testing checks that an implementation fulfills its specification. Several authors [8, 14, 30] have dealt with the problem of *automatically generating test suites to test the conformance of an implementation under test (IUT) to its specification*, where both specifications and IUT are expressed in the form of Labeled Transition Systems (LTS), Finite State Machines (FSM) or Input/Output LTS (depending on the approach).

Some interest has been devoted to *Software Architecture-based conformance testing* [6, 7]. Given a software architecture (SA) description, conformance testing has been used to detect conformance errors between the SA specification and its implementation. The SA specification has been used as a reference model to which the source code should conform.

One of the most challenging problems of SA-based conformance testing is the necessity of a common model that makes it possible to compare the expected behavior of a SA with its real implementation. A common model would bridge or map the elements from these two different abstractions, addressing the so called "mapping problem" or "traceability problem". Traceability concerns "relating the abstract values of the specification to the concrete values of the implementation" (as quoted from [12]). Several researchers have recognized the importance and difficulty of this step [12, 32, 27], which has been deeply analyzed in [6].

The main goal of this paper is to review and extend our previous work on SA-based conformance testing, to provide a ***systematic way*** to use an SA for code testing. This research is driven by a previous analysis we performed in [7], where we identified the factors making the distance between SA and code high. As a result, the development process, the relationships among architectural components and the source code, and the SA-level of abstraction strongly influence that distance. If "explicit mapping rules (could) drive the source-code implementation from architectural components, connectors, and messages" [7], then the mapping problem could be easily managed.

In this paper, we propose a specialization and refinement of our general approach for SA-based conformance testing, in order to obtain a systematic approach for performing code level conformance testing based on SA specifications. Here, we deal with this problem in a specific SA style, the C2 style [29, 10]. We show how the SA to code mapping rules imposed by the C2 framework helps to limit the mapping problem, and allows a systematic testing approach.

This research is part of a project called SARTE (SA-based regression testing) [22], which aims to provide an approach and framework to test software architectures and code, when both are subject to changes.

The rest of the paper is structured as follows: Section 2 provides an overview on Software Architecture-based Testing. Section 3 describes the research outline and our proposal. Section 4 describes the C2 architectural style and presents the case study used in this paper. Section 5 explains how the general approach mentioned in Section 2 can be specialized to C2 style architectures. Some results from our experiment are presented in Section 6. We discuss some related work in Section 7, and our conclusions and future work in Section 8.

2 Software Architecture-Based Testing: An Overview

The topic of *architectural testing* has recently raised some interest [25, 4, 16, 27, 7].

In [25], the authors define six architectural-based testing criteria, adapting specification-based approaches; in [4] the authors analyze the advantages of using SA-level testing for reuse of tests and to test extra-functional properties. In [16] the author presents a discussion on the use of software architecture for testing. In [27], the authors present an architecture-based integration testing approach that takes into consideration architecture testability, simulation, and slicing.

The approach proposed in [7] is, to the best of our knowledge, the first effort to tackle the whole cycle of SA-based testing with a comprehensive and concrete approach. It spans the spectrum from test derivation based on architecture dynamics down to test execution over system implementation.

The general approach is composed by some logical steps which can be outlined with the help of Figure 1.

In *Step 0*, a topological and behavioral specification of the SA is required.

In *Step 1*, a software architect, by looking at the software architecture dynamics from different viewpoints, defines various testing criteria [5]. Each criterion highlights a specific perspective of interest for a test session and is realize through an *obs-function.*

Step 2 derives, through the selected obs-function, an Abstract LTS (ALTS), which still expresses all high-level behaviors we want to test, but hides any other irrelevant behavior.

Step 3 uses the ALTS in order to select an architecture-level test suite. Each "architectural test" is a sequence of architecture-level actions meaningful with respect to the testing criterion.

Finally, *Step 4* uses the architectural test cases to actually test whether the source code conforms to the architectural description. This step has to identify how SA-level abstract test cases can be related to concrete values of the implementation (i.e., *traceability/mapping* among SA and code) and how the code may be run over the identified test cases. The traceability problem has been handled by using an informal "mapping" function while the execution traces are analyzed to check whether the system implementation works correctly with respect to the architectural tests.

The goal of this paper is to improve and refine that work, in order to handle, in a systematic way, all the testing approach, as outlined in Section 3.

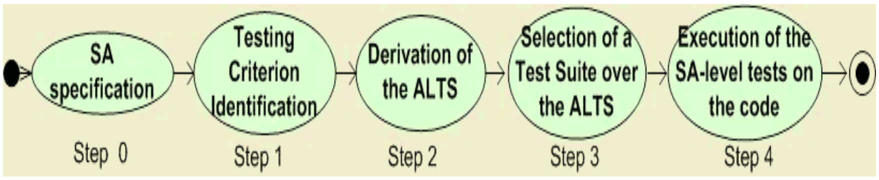

SA specification
Testing Criterion Identification
Derivation of the ALTS
Selection of a Test Suite over the ALTS
Execution of the SA-level tests on the code
Step 0
Step 1
Step 2
Step 3
Step 4

Fig. 1. An Activity Diagram of a SA-based Testing.

3 Research Outline

The SA-based testing process proposed in [7] is largely mechanical, but some important human interventions are required.

The *test selection stage* (steps 0 to 3) is systematic: the SA is formally specified using an ADL of a model-based specification, the LTS is automatically generated from the specification, the ALTS can be automatically generated using existing tools, the SA test selection process is implemented using existing tools to cover the ALTS.

The *test execution stage* (step 4), instead, is informal and left to the software engineer ability. In particular, a software engineer has to manually deal with the traceability problem, i.e., "relating the abstract values of the specification to the concrete values of the implementation" [12]. This important problem has been already recognized by other researchers [32, 27] but never formally handled.

In previous papers [6, 7], we managed such problem in a very general context, supposing that a well-formalized architecture-based development process was ***not*** in place (as happens in real world) and the SA specification and the low-level design have been intermixed without any formal mapping. One advised solution was to use some development support which explicitly adopts a formal mapping between architectural and implementation elements.

In this paper we specialize and refine some of the activities presented in Figure 1. We choose a specific architectural style, the C2 style, which supports implementation and facilitates the mapping among SA-based and code-based test cases. We enrich the C2 structural specification with a behavioral one, to accommodate a behavioral model of the system. We reuse and adapt previous experience to identify and select relevant architectural test cases. In particular, we completely revise steps 3 and 4.

We use existing tools (namely, Argus-I [2]) in order to run deterministic testing. We thus apply this technology to the Elevator case study, described in Section 4.2.

4 Preliminary Information

In this section we provide information which will be useful in the following. We outline the C2 style for describing software architectures and we present the Elevator case study, used in Section 5 to put in practice the proposed approach.

4.1 C2 Style Software Architectures

C2 [10, 29] is an architectural style introduced in 1995 by researchers from the University of California, Irvine. This style imposes some compositional and behavioral rules [29] enabling some level of independence (called "substrate independence") between the components used to describe the SA.

Components have visibility only on components up to them but they do not need any information on components beneath them. Moreover, communication may happen only through the explicit use of connectors. Each component and each connector exposes exactly two interfaces, to send "requests" and receive "notifications"; a request consists in requiring a service to other components while a notification identifies the output of a request. "Links" are used to configure a C2 style architecture, by relating component and connector interfaces. C2 components, connectors, interfaces and links can be visualized and analyzed using, for example, the Argus-I [2] tool.

The *C2 framework* [10] helps software engineers to produce the actual implementation of the architecture. In the following, we assume our SA complies to the C2 style and the implementation is realized through the C2 framework.

4.2 The Elevator Case Study and Its Software Architecture

Elevator systems have been widely used in testing and analysis literature because of two main reasons: everyone is familiar with elevator systems, and can easily understand the requirements for such application domain; and these systems contain concurrent, stateful components and timing requirements, which give them a level of complexity that is interesting for verification purposes.

In the configuration for our case study, the elevator system contains the *building panel* (which includes all the panels from different floors of the building), two *elevator cars,* and a *scheduler* algorithm to assign calls requested through the building to the closest elevator car.

The components of this elevator system are:

- ***ElevatorADT*: this component maintains the information about the elevator car state, such as: motion {moving, stopped-closed, stopped-opened} and direction {up, down}. In addition to state information, the ElevatorADT keeps a list of all the calls it needs to attend.
- ***ElevatorPanel*: this component represents the internal panel of an elevator car. After entering the elevator, the passenger can request calls through it, and see the current floor.
- ***BuildingPanel*: this component represents all the elevator call panels of the building. Through this component, users in different floors can request a call to the elevator system, indicating the desired direction.
- ***Scheduler*: this component receives call requests from the BuildingPanel, and selects which elevator should attend such call. In our case study we are using a "runtime" scheduling policy so that if a call is made at time "t", it initially selects the elevator car (EC) that, at time "t", could attend it with the lower waiting time required. At time "t+i", i = {1, …, n}, the Scheduler checks if EC is still the best option, and, if it is not anymore, can switch to another elevator. This process is repeated until the call is served by one elevator.
- ***Synchronizer*: this component synchronizes the elevator movements, so that all of them makes a move at the same time.

5 Systematic Testing of C2 Style Architectures

In this section we describe how the generic SA-based conformance code testing proposed in [7] and outlined in Section 2 can be instantiated to C2 style architectures in order to become systematic. The theory is thus applied to the Elevator example described in Section 4.2.

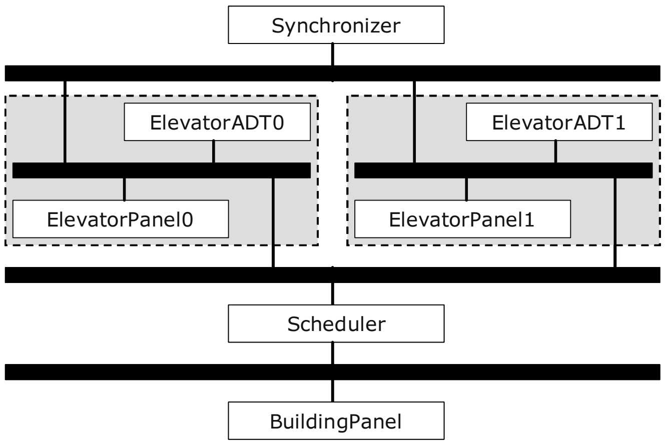


Fig. 2. The Elevator C2 style architecture.

5.1 Step 0: SA Specification

The system software architecture is modeled following the C2 style. Figure 2 shows the Elevator architecture, as visualized by Argus-I [2]. Many requests and notifications are sent and received by components, through the five connectors shown in Figure 2 (thick horizontal lines). Such requests are used to realize different services, like *AddCall*, *CallAttended*, *GetId*, *Suspend Clock*, *Resume Clock*. (The description of the C2 architecture, complete of requests and notifications, may be found in [23], Appendix A).

It is to be noticed that the C2 specification is just structural, and a behavioral description of components and connectors interaction is missing. Some work has been done in the past on this direction, formalizing a portion of C2 in the Z specification language [19] or by using pre- and post-conditions, but they are still really preliminary. We thus decided to complement the C2 style specification with a behavioral one. We use the Finite State Process (FSP) [18] algebra. In this notation, each component and each connector is represented through a process. The behavior of each process can be automatically represented through a Labeled Transition System (LTS) and those LTSs can be combined together (following the FSP semantics of composition) in order to produce a global LTS, describing how components and connectors work together. We choose this language, since it is tool supported (LTSA tool [17]) and we had previous experience with it. We modeled the behavior of each single Elevator component in FSP. The FSP specification for the Elevator architecture can be found in [23], Appendix B.

By using the LTSA tool over the FSP specification, an LTS has been generated for each component. Such LTSs have been combined together generating a global LTS composed by 4288 states and 26336 transitions (before minimization), representing how the system evolves when different services are requested or notified.

This model will be used in the following to extract SA-level test cases.

5.2 Step1: Definition of an Observation

In principle, the global LTS could directly be used as the reference model for deriving the test scenarios, assuming that an architectural test is essentially a *sequence of system actions meaningful at the architectural level*. Unfortunately, by considering this global graph, it is very difficult to realize the testing *selection stage* [7], i.e., to "suitably" select a set of test cases. In fact, the LTS provides a tremendous amount of information flattened into a graph (i.e., many messages, services, parallelism, interleaving, many components interacting and so on). By identifying some *testing criteria* [5], we can select only such behaviors suitable for the criterion itself, abstracting away uninteresting behaviors. We can thus focus on a subset of relevant behaviors.

An *obs-function* (as defined in [7]) which partitions LTS actions into "relevant interactions" *R* (i.e., those we want to observe by testing) and "non-relevant interactions" *NR* (i.e., those we are not interested in at this time) can realize such a testing criterion.

In the context of a C2-FSP specification, we extend the concept of *obs-function* into *structural* and *functional* observations. A structural observation focuses on the SA topological description provided by C2. The software tester, looking at the C2 specification, identifies components/messages/connectors she is more interested to test. A functional observation, instead, tries to capture relevant information from the behavioral model. With a functional observation, the tester can identify system functionalities she is more interested to test.

In this paper, due to space limitations, we describe one functional observation. The functional observation of interest is "*all those behaviors involving the **AddCall** and **CallAttended** services*". In other words, we consider as "relevant" all and only, those interactions necessary to realize the *AddCall* and *CallAttended* services, while hiding the others. In the following, we refer to this testing criterion as the "AddCall+CallAttended" *obs-function.*

5.3 Step 2: Derivation of the Abstract LTS

An *obs-function* allows to derive an Abstract LTS (ALTS) which satisfies the criterion itself and still expresses all high-level behaviors we want to test in the initial LTS.

The AddCall+CallAttended observation has been produced by modifying the Elevator FSP specification, hiding all such events not relevant for our testing criterion. The new FSP specification produces an ALTS composed by 41 states and 51 transitions. This ALTS reduces the initial LTS by describing those requests and notifications related to the *AddCall* and *CallAttended* services only.

It is to be noticed that the hiding operator does not delete the components from the specification. It just makes invisible the messages sent and received by the hidden components, still guaranteeing the model correctness.

5.4 Step 3: Selection of a Test Suite over the Observed Behavior

In code-based testing, a test case is usually defined as the input value provided to a program P, with the corresponding expected output. At the architecture level, a test case can be defined as:

Definition*: *SA Test case
An SA test case is an ordered sequence of architectural events observed when a certain initiating event is performed.

This definition encompasses two different keywords: the sequence of actions, which represent expected behaviors, and the initiating event, that is, the architectural input which should allow the sequence to happen.

The expected sequence of actions, for a certain testing criterion, can be extracted by applying a path coverage over the ALTS. Each ALTS path represents a sequence of expected architectural events. Many coverage criteria can be applied. A *complete path coverage* criteria can be applied when the ALTS dimension is reasonable, and when we are interested in a thorough coverage. In [7] we proposed to use *McCabe's path coverage* criteria [31] in order to provide a less thorough coverage by identifying only independent paths.

In this paper, we adapt the ***category partition method*** [24]. Given a functional unit of interest, the category partition method requires to identify functional "*parameters*" and "*environment conditions*". A parameter is an input to the functional unit while the environment condition is a characteristic of the system's state at the execution time. Following the category partition method, mutually exclusive "*choices*" are identified for each parameter and condition (i.e., parameters and conditions values), "*constraints*" are identified for each choice (i.e., when a choice may occur) and "*test frames*" are identified by computing the cross-product of the different choices (i.e., choices are combined together).

Given the ALTS for the *AddCall+CallAttended* obs-function previously defined, we applied the category partition method in order to select ALTS paths of interest. The *AddCall* service represents the functional unit of interest. The only input this service receives is the "add call" request from the Building Panel. There are no choices or constraints related to this input. However, there are many environmental conditions to be considered, as reported in Table 1 and described in the following:

- The *Scheduler* ***selects*** which elevator should attend the call, based on the lower waiting time required. ADT0 or ADT1 can be selected to handle the call (condition #1, Table 1);
- The *Scheduler* can ***check again*** which is the best elevator for the call, depending on a periodical check (condition #2, Table 1);
- The *Scheduler* can ***reselect*** the elevator, if the check again condition is true. If the initial choice is still the best, the initial elevator attends the call, otherwise, there is a switch to another elevator (condition #3, Table 1).

Table 1 describes the environmental conditions, with the possible choices and constraints. Table 2, instead, reports the test frames we are interested to test. Test frame #1, for example, means that we are interested to test the following behavior: the elevator ADT0 is selected initially and a reselection process does not happen.

Table 1. Environmental Conditions, choices and constraints.

#	Environment Condition	Choice	Constraint
1	*Elevator selection (ES)*	ADT0 or ADT1	Lower waiting time
2	*Check again (CA)*	YES or NOT	Periodical check
3	*Elevator reselection (ER)*	ADT0 or ADT1	Lower waiting time

Since the conditions, their choices and constraints were selected for the AddCall and CallAttented services, a mapping among the ALTS paths and the test frames is always possible. In our example, 144 complete paths were extracted from the selected ALTS and partitioned into the six test frames. Table 2, column three, says that 8 different paths satisfied test frame #1 and #4, while 32 paths satisfied the others (for a total of 144 different paths). Reusing the idea proposed in the category partition method, we can select one ALTS path for each test frame, as representative of the all set. The six representative paths are listed in [23], Appendix C.

Table 2. Test Frames and ALTS paths.

#	Test Frames	# of ALTS paths in the partition
1	ES = ADT0, CA = NOT	8
2	ES = ADT0, CA = YES, ER = ADT0	32
3	ES = ADT0, CA = YES, ER = ADT1	32
4	ES = ADT1, CA = NOT	8
5	ES = ADT1, CA = YES, ES = ADT1	32
6	ES = ADT1, CA = YES, ES = ADT0	32

5.5 Step 4: Tests Execution over the Source Code

This step describes how i) SA-level abstract test cases can be related to concrete values of the implementation (i.e., *traceability/mapping* among SA and code) and how ii) the code may be run over the identified test cases. We analyze those two distinct topics in the following subsections.

Refinement of the Architectural Tests into Code-Level Tests

One of the reasons we decided to start this research using the C2 style, instead of a generic SA, is that C2 is supported by the C2 framework [10], which dictates how C2 style architectures have to be implemented. The C2 framework can be considered as a set of predefined abstract classes and interfaces that have to be implemented following certain constraints when developing a C2 style architecture. The framework allows the software engineer to implement a C2 style architecture in a straightforward manner: each architectural component is implemented by a Java component. Auxiliary classes can be introduced in order to implement specific aspects. SA events have exactly the same signature in the code and in the FSP architectural specification, the mapping is one-to-one based on signature matching.

Thanks to the strong relationship among a C2 specification and its implementation, the mapping between architectural test cases and code-level test cases may be performed systematically. In order to test the code conformance to a selected SA test case, we could run the code, make an elevator call and check if one of the architectural test cases is traversed. However, depending on the system status (e.g., elevators floor and direction, call made in a specific floor to go up or down), all the 144 ALTS paths, in the six test frames, could be executed. This means that we need to refine the parameters and environment conditions previously identified in order to use an SA test case as an oracle.

Let's see how the refinement process may work, by using the ALTS path in Figure 3 (path #1 in [23], Appendix C). This path is representative of the test frame #1 in Table 2. We are interested to test if this specific behavior can be executed at the code level, when the constraints identified in test frame #1 are verified, that is, *assuming that ADT0 has the lower waiting time* (ES = ADT0) *and the periodical check does not apply* (CA = NOT). In order to refine path #1 (as any other path), we identified execution parameters which allow *ADT0 to have the lower waiting time* and *the periodical check not to apply*. We found out that:

- the waiting time constraint depends on the *direction* and current *floor* of the two elevators, on the floor the call is made, and on the direction the user wants to go;
- the periodical check does not apply only when the building panel and the elevator ADT0 are at the same floor and the call has the same direction of ADT0.

It means that path #1 has to be executed when BP(x,y), ADT0(x,y), ADT1(*,*) holds, that is, both BP and ADT0 are at the same floor "x", BP makes a call to go *up/down* when ADT0 is going in the same direction "y" and ADT1 may be in a generic floor with a generic direction.

Table 3 shows seven code-level test cases. Test case 7, for example, states that when the AddCall is sent, if both BP and ADT0 are at the third floor going up, ADT0 should be selected to get the call and the ALTS path #1 should happen.

Summarizing, the idea is to reproduce the initial condition so that the architectural test case should happen. This is not in general an easy step, but as shown in the next paragraphs, it is made mechanic thanks to the use of Argus-I.

Tests Execution

Our pragmatic approach here is to make a *deterministic* [9] analysis of the code execution to observe the desired sequence. The deterministic approach forces a program to execute a specified test sequence by instrumenting it with synchronization constructs that deterministically reproduce the desired sequence. This determinist analysis is performed through monitoring and debugging capabilities provided by Argus-I [2], by setting breakpoints during code execution.

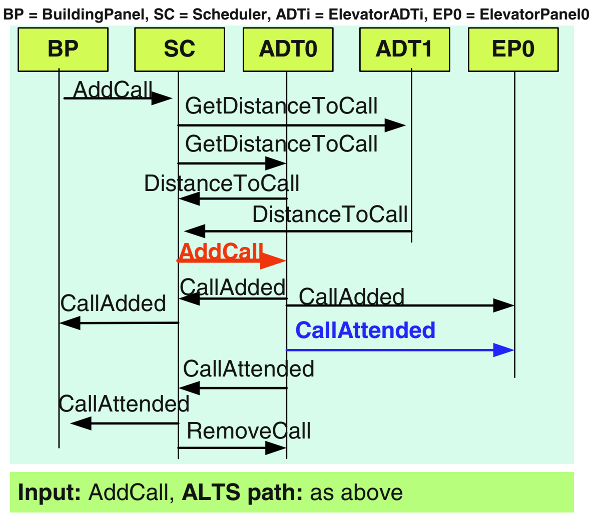


Fig. 3. Architectural Test Case for the AddCall+CallAttended observation.

Table 3. Mapping the SA path 1 to low level test cases.

Test Path	Test Case
Path 1	1) BP(1,up) ADT0(1,up) ADT1(1,up)
	2) BP(5,down) ADT0(5,down) ADT1(5,down)
	3) BP(5,down) ADT0(5,down) ADT1(5,up)
	4) BP(10,down) ADT0(10,down) ADT1(5,down)
	5) BP(1,up) ADT0(1,up) ADT1(2,up)
	6) BP(3,up) ADT0(3,up) ADT1(2,up)
	7) BP (3,up) ADT0(3,up) ADT1(4,down)

We force the system to be in a state described by the test case (as in Table 3), then we try to deterministically force the execution of one of the ALTS paths compliant with the test frame.

When it is not possible to reproduce one of the expected architectural behaviors, the system implementation is not behaving as expected, i.e., it is not conform to the architecture specification. In such cases, an architectural error is revealed.

6 Results and Considerations

We used the C2 framework in order to produce a Java implementation of the Elevator system. Moreover, we produced a faulty version of the same system, injecting some faults. We thus tested the two systems for conformance to the six architecture level test cases selected in Section 5.2 (also listed in [23], Appendix C).

Table 4 summarizes relevant results. We used 42 (code level) test cases over the two versions of the code (V1 is the faulty implementation while V2 is the initial one). SA level test cases #3 and #6 (related to the test frames #3 and #6 in Table 2) detected many faults in the code faulty version V1. These errors are due to the fact that, when a call request R is rescheduled from one elevator (A) to the other (B), while elevator B receives the AddCall(R) event, both elevators receive the RemoveCall(R) event. Therefore, since elevator B received a RemoveCall(R) just after the AddCall(R), it does not attend call R, which is left unattended by the elevator system.

More importantly, SA level test case #4 (related to the test frame #4 in Table 2) detected a "real" error in both the original and faulty version. Basically what happens is that although elevator ADT1 was supposed to receive the AddCall event, elevator ADT0 is the one actually receiving it.

Table 4. Results.

SA test case #	# of Code level test cases	V1 Faulty Implementation	V2 Original Implementation
1	7	No faults detected	No faults detected
2	8	No faults detected	No faults detected
3	6	**5 faults detected**	No faults detected
4	7	**1 fault detected**	**1 fault detected**
5	8	No faults detected	No faults detected
6	6	**5 faults detected**	No faults detected

In order to produce unbiased results, we performed this evaluation separately: one of the authors produced the SA specification of the system and the SA-level test cases while the other implemented the system, refined the SA test cases into code level test cases and run the test cases.

The first consideration to be done is that C2 is used in this paper as representative of all such frameworks which support a code generation process. The interest of this research, in fact, is to analyze how a generic framework, supporting the code generation, may help to make systematic the testing process. Moreover, the assumption that the code generation process is driven by a superimposed framework is not to be considered too restrictive. Recent research is investigating how ADLs can support the generation of executable code [20, 25]. In particular, in [25] the authors analyze how ADLs are evolving in order to bridge the gap between a software architecture specification and its implementation. Both Monroe [21] and Garlan [15] point out how skeletal code automatic generation may reduce implementation time. Moreover, Aesop, C2, and Darwin generate skeletal code in C/C++ and Rapide executes the design code internally. Furthermore, MetaH is supported at the implementation level by Ada, and ArchJava. For more details, please refer to [25].

It may be of interest, for future research, to analyze how this experience may be reused in contexts different from C2. We expect that when the code generation process is systematic, the testing process may be performed systematically too.

Some problems we initially found were how to run the code and deterministically analyze the code execution and how to identify parameters. Thanks to the Argus-I tool we overcame the first problem. Following the category partition method, we easily learned how parameters may be identified.

7 Related Work

In this section we briefly present important research areas related to our approach.

The topic of ***specification-based conformance testing*** has been extensively analyzed by many authors [8, 30, 14] , as already pointed out in the Introduction. Comparing our approach with their, we can notice that we also use the SA-derived LTS as a reference model to derive test cases. However, all such approaches produce a model of the implementation under test and define some implementation relations (conf, ioconf, ioco, etc.) between code and specification. In our case, we do not assume to be able to produce an LTS model of the implementation thus we compare architecture level sequence of events with lower level execution paths.

The topic of ***SA-based testing*** has been already discussed in Section 2. Again, the main difference between our and other approaches is the challenge to consider the whole cycle of SA-based testing, from architecture specification to test execution over system implementation.

The difficulty of ***tracing*** information is not new, as already recognized in [12, 32, 27]. Some relevant papers have been written on the topic. We can here mention reference [13] which shows a way to detect traceability between software systems and their model and proposes a list of interesting references on traceability techniques. Some work has been done in bridging the gap between requirements and software architectures (e.g., [28]), and much other work addresses requirements traceability. The problem of mapping abstract tests into the System Under Test is under study in the ongoing AGEDIS project [1].

8 Conclusions and Future Work

In this paper we refine our previous experience [6, 7] on SA-based testing. While our previous papers were dealing with a generic architecture in a generic software development process (without assuming any relationship between SA and code), we here make a stricter assumption on the software development process, in order to make more formal the full testing process.

In particular, by adopting the C2 style architecture and the related C2 framework, we are able to systematically implement the *test execution stage* described in step 4, handling both traceability/mapping among SA and code execution over the identified test cases problems. By using C2, in fact, traceability is explicitly maintained between the architectural events and the code-level sequences while code execution is allowed through the Argus-I tool.

In future work we want to investigate how a similar approach can be applied to those other ADLs which support code generation. Moreover, we want to apply this approach for SA-based regression testing, in the SARTE project [22].

In the long term, we plan to use the experience gained in this paper in order to relief some constraints. Our desire is to be able to specify and test architectures in a generic ADL (assuming that a behavioral model can still be produced). In particular, we would like to take a generic architecture described using the XADL ADL [11], providing a behavioral description in the form of state-based machine model, to implement this architecture using a component-based technology through a middleware (Java/RMI, COM+ of CORBA) and test the system implementation.

Acknowledgments

The authors gratefully acknowledge the Italian MIUR national project SAHARA and the National Science Foundation, both of which partially supported this work.

References

1. AGEDIS Project. Automated Generation and Execution of Test Suites for Distributed Component-based Software. On-line at: *http://www.agedis.de/index.shtml*.
2. The Argus-I project. University of California, Irvine. Information on-line at *http://www.ics.uci.edu/~mdias/ research/ArgusI*.
3. G. Bernot, M. C. Gaudel, and B. Marre. Software Testing Based on Formal Specifications: A Theory and a Tool. Software Engineering Journal, Vol. 6, N. 6, pp. 387-405, 1991.
4. A. Bertolino and P. Inverardi. Architecture-based software testing. In *Proc. ISAW96*, October 1996.
5. A. Bertolino. Knowledge Area Description of Software Testing. In *SWEBOK*: The Guide to the Software Engineering Body of Knowledge, Joint IEEE-ACM Soft. Eng.
6. A. Bertolino, P. Inverardi, and H. Muccini. An Explorative Journey from Architectural Tests Definition downto Code Tets Execution. In IEEE Proc. Int. Conf. on Software Engineering, ICSE2001, pp. 211-220, May 2001.
7. A. Bertolino, P. Inverardi, H. Muccini. Formal Methods in Testing Software Architectures. Chapter in Formal Methods for Software Architectures, SFM-03: SA Lectures, Eds. M. Bernardo, P. Inverardi, LNCS 2804, 2003, p. 124-149.
8. G. v. Bochmann and A. Petrenko. Protocol Testing: Review of Methods and Relevance for Software Testing. In ACM Proc. Int. Symposium on Software Testing and Analysis, ISSTA '94, pp. 109-124, 1994.
9. R. H. Carver and K.-C. Tai. Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs. *IEEE Trans. on Software Engineering*, Vol. 24, N. 6, pp. 471-490, June 1998.
10. The C2 Architectural Style. On-line at: *http://www.ics.uci.edu/pub/arch/c2.html*.
11. E. Dashofy, A. van der Hoek, and R. N. Taylor An Infrastructure for the Rapid Development of XML-Based Architecture Description Languages. In Proceedings of the ICSE 2002 International Conference on Software Engineering (ICSE 2002), Orlando, Florida, May 2002.
12. J. Dick and A. Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In J.C.P. Woodcock and P.G. Larsen (Eds.), *FME'93: Industrial-Strenght Formal Methods*, pp. 268-284. *LNCS 670*, 1993.
13. A. Egyed. A Scenario-Driven Approach to Traceability. IEEE Proc. Int. Conf. on Software Engineering (ICSE2001), pp. 123-132, May 2001.
14. J.-C. Fernandez, C. Jard, T. Jeron, L. Nedelka, and C. Viho. Using On-the-fly Verification Techniques for the Generation of Test Suites. In *Proc. of the Eighth Int. Conf. on Computer Aided Verification* (CAV'96), USA, pp. 348-359, 1996. *LNCS 1102*, Springer, 1996.
15. D. Garlan. "Software Architecture", *Encyclopedia of Software Engineering*, John Wiley & Sons Inc., 2001.

16. M. J. Harrold. Testing: A Roadmap. In A. Finkelstein (Ed.), *ACM ICSE 2000, The Future of Software Engineering*, pp. 61-72, 2000.
17. Labelled Transition System Analyzer (LTSA). On-line at: <*http://www-dse.doc.ic.ac.uk/~jnm/book/*>.
18. J. Magee, and J. Kramer. *Concurrency: State models & java programs.* Wiley publisher, April 1999.
19. N. Medvidovic. Formal definition of the Chiron-2 software architectural style. Technical Report UCI-ICS-95-24, Department of Information and Computer Science, University of California, Irvine, August 1995.
20. N. Medvidovic, and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. In IEEE Transactions on Software Engineering, vol. 26, no. 1 (January 2000).
21. R. T. Monroe, Rapid Development of Custom Software Architecture Design *Environments*, Ph.D. Thesis, Carnegie Mellon University, 1999.
22. H. Muccini, M. Dias, and D. Richardson. Software Architecture-based Conformance and Regression Testing - documents. On-line at: *http://www.HenryMuccini.com/Research/ICSE04_Submitted.htm.*
23. H. Muccini, M. Dias, and D. Richardson. Systematic Testing of Software Architectures in the C2 style. Extended version of the ETAPS 2004 publication. On-line at: *http://www.HenryMuccini.com/Research/ETAPS04.htm.*
24. T. J. Ostrand and M. J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of the ACM*, Vol. 31, N. 6, pp. 676-686, June 1988.
25. O. Papapetrou, A. Stavrou and G. A. Papadopoulos. From Software Architectures to Coordination Programming: Bridging the Gap Between Architecture Specification and System Implementation. Submitted for publication.
26. J. Richardson and A. L.Wolf. Software testing at the architectural level. *ISAW- 2* in Joint Proc. of the *ACM SIGSOFT '96* Workshops, pp. 68-71, 1996.
27. J. Richardson, J. Stafford, and A. L. Wolf. A Formal Approach to Architecture-based Software Testing. Technical Report, University of California, Irvine, 1998.
28. Straw 2001. First Int. Workshop "From Software Requirements to Architectures" (STRAW'01), May 14, 2001, Toronto, Canada.
29. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. In IEEE Transactions on Software Engineering, June 1996.
30. J. Tretmans. Conformance Testing with Labeled Transition Systems: Implementation Relations and Test Generation. *Computer Networks and ISDN Systems*, Vol.29, pp. 49-79, 1996.
31. H. Watson and T. J. McCabe. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric.* NIST Special Publication 500-235, August 1996.
32. M. Young. Testing Complex Architectural Conformance Relations. In *Proc. Int. Workshop on the ROle of Software Architecture in TEsting and Analysis* (ROSATEA), CNR-NSF, pp. 42-45, July 1998.

Optimising Communication Structure for Model Checking

Peter Saffrey[1] and Muffy Calder[2]

[1] Dept. of Computer Science, University College London
P.Saffrey@ucl.ac.uk
[2] Dept. of Computing Science, University of Glasgow
muffy@dcs.gla.ac.uk

Abstract. Model checking is an effective tool in the verification of concurrent systems but can require skillful use. The choice of representation for a particular system can make a substantial difference to whether the verification will prove tractable. We present a method for improving the choice of representation by effective use of *communication structure*. The main contribution is a technique for selecting a communication structure which yields a reduced search space whilst preserving the essential behaviour of a representation. We illustrate our method with examples based on the model-checker Spin.

1 Introduction

Concurrent systems consisting of a number of communicating processes are present in many real world applications. However, the complexity inherent in communication and parallelism makes it difficult to build concurrent systems that behave as intended without errors or failures.

One technique to aid in the construction of reliable concurrent systems is *model checking* [2]. Model checking attempts to *verify* the behaviour of a system by exploring all possible behaviours of that system, the *state space*, by checking each behaviour against a set of *properties* which are expected to hold, or be violated. This procedure can be expensive and for some systems the state space may be too large for a complete search: the verification is thus intractable.

When verifying a real world system using model checking, a particular *representation* (i.e. a model) of that system must be chosen. It is usually possible to choose a variety of representations, any one of which would accurately represent the behaviour of that system.

An illustration of this phenomenon is shown in figure 1. Each representation consists of a model and an associated set of specified properties. Each representation results in a state space, often of differing size.

Choosing, or developing a representation for a system is analogous to the act of converting a specification into a piece of software, often called *programming*. In most cases, a large number of programs can be written to conform to a single specification, and they may vary widely in time efficiency, space efficiency or other measure of quality.

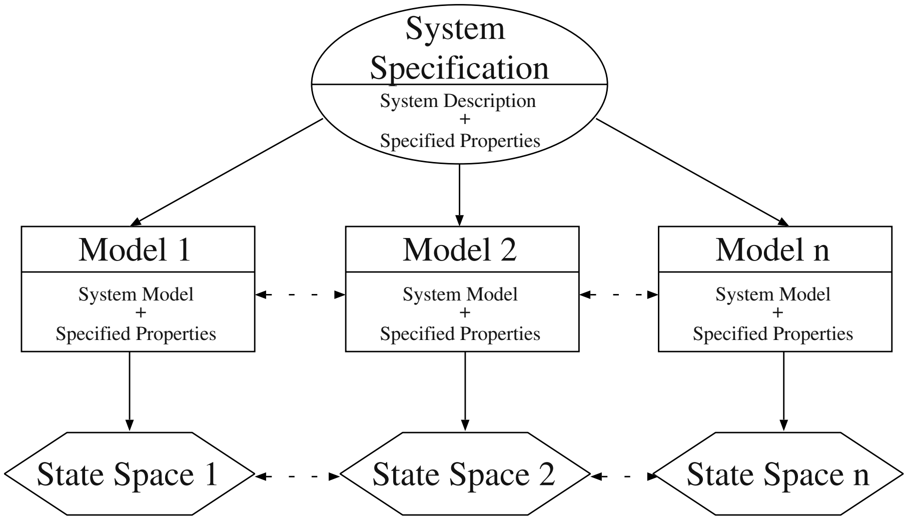

System Specification
System Description + Specified Properties
Model 1
System Model + Specified Properties
Model 2
System Model + Specified Properties
Model n
System Model + Specified Properties
State Space 1
State Space 2
State Space n

Fig. 1. Alternative models and their state spaces.

Although these observations are relevant to all aspects of representing a system for model checking, we will apply these notions to one specific area: that of *communication structure*. In a concurrent system, the communication structure is the method by which information passes between components. More precisely, it refers to the data structures used for communication, and which processes may access those data structures (to read or write). Specification languages for model-checkers provide a number of constructs to represent communication structure. How these constructs are used can have a considerable influence over the size of the resulting state space.

Communication structure is most pertinent to asynchronous communication since in synchronous systems, messages are transmitted instantaneously and thus the choice of structure has little impact on the state-space. We therefore concentrate on asynchronous communication and the use of *channels*. The focus of this work is not generic, internal data structures for model checking, but rather the specific, source language data structures used in problem representation. The inspiration to explore the relationship between communication structure representation and state space was provided by [6] in which the author demonstrated a reduction in state space for an example based on the logical linked protocol.

The main contribution of this paper is a technique for selecting a communication structure which preserves the essential behaviour of a representation and yields a tractable search space. We assume the starting point is an intractable initial representation. Therefore, our technique must be applicable *without* model checking the initial representation.

The remainder of the paper is organised as follows. We open with a motivating example before justifying the model checker we have chosen to illustrate our technique (sections 2 and 3). Sections 4, 6, 7, 8 and 9 outline our technique for demonstrating property preservation between communication structures and how good communication structures can be chosen. Section 10 provides a case study. Finally, sections 11, 12 and 13 provide related work, further work and conclusions.

2 Communication Structure

To illustrate how communication structure can influence the size of a resulting state space, we present a simple example.

Consider the system shown in figure 2 with its two alternative communication structures. Two users may send messages to one another along communication channels. In the first case, two *dedicated* channels are used, a channel for each direction. In the second case, communication in either direction is mediated by a single *shared* channel.

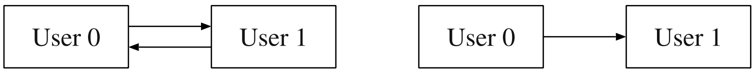


Fig. 2. A simple system with alternative communication structures.

Assume that each user can only send one type of message and that each channel can contain a maximum of one message. Table 1 enumerates the message combinations for the shared and dedicated configurations: [] denotes all channels empty, (user0,user1) a message from user0 to user1 and (user1,user0) a message from user1 to user0. Note that a state where both messages are being passed simultaneously is only possible for the dedicated structure; its state space size is greater by 1.

Table 1. Enumeration of combinations for simple system.

Dedicated	Shared
[]	[]
(user0,user1)	(user0,user1)
(user1,user0)	(user1,user0)
(user0,user1), (user1,user0)	—

The simple system can be extended to three users, where the dedicated structure has 6 channels (2 between each user) and the shared structure 3 channels (1 between each user). Here the dedicated structure has 64 possible states while the shared structure only 27. As the topology becomes more complex, the difference in the number of combinations which can achieved with various communication structures tends to increase.

It is clear that a change from a dedicated to a shared structure alters the size of the state space in this simple example. However, these systems are not isomorphic: one state is only reachable with the dedicated structure. There are also differences in the possible series of events that that can occur. For example, in the dedicated system, it is possible for a send event from user0 to immediately follow a send event from user1; in the shared configuration, there must be a receive event to clear the channel.

This example demonstrates how a change to communication structure can affect state space size but with a loss of certain behaviour. Whether such differences in behaviour influence the verification of specified properties for this system is a primary question addressed by our work. A further question is which representation will result in the smallest state space. We will consider some guidelines for selecting a representation later, in section 9. For now we concentrate on the more problematic issue of altering a representation, i.e. traversing the horizontal arrows in figure 1. For alteration to be applicable, we must provide some demonstration of equivalence between the initial and the altered system.

3 Spin and Promela

Our method is designed to apply to any systems that are modelled with asynchronous communication. To validate the techniques we have constructed an implementation based on the model checker Spin [5] and its accompanying specification language Promela. Spin applies state of the art, on the fly, model checking techniques. Promela is a succinct and easy to use language that supports the use of channels for communication. All the examples presented in this paper were represented in Promela and model checked using Spin.

4 Communication Structure Alteration

In this section, we address communication structure alteration: altering the communication structure of a representation to result in a smaller state space.

Communication structure alteration takes as input an *initial representation* with an initial communication structure. The aim is to produce an *alternative representation* which results in a smaller state space and which preserves the *specified* properties of the initial representation. It is important to note that only the specified properties must be preserved, other properties might change.

The procedure follows the following stages:

1. The initial representation is analysed including the extraction of the communication structure.
2. Using a best communication structure framework (see section 9), a new communication structure is selected that results in a smaller state space. The initial representation with this new communication structure is referred to as the *candidate representation*.
3. The initial and candidate representations are compared to test for property preservation. Details of this procedure are in section 6. After comparison,
   – If the candidate representation preserves the properties of the initial representation, the candidate representation, with its smaller resulting state space, can be used to verify specified properties of that system.
   – If alteration is not property preserving, return to stage 2 and choose another communication structure.

A communication structure alteration only alters the communication structure: no other part of the system – for example, the number of components, the behaviour of those components or which components interact with each other – is altered.

Communication structure alteration should only be applied when the specified properties can be preserved by the alteration. Our method for testing whether or not properties are preserved is described in section 6. Recall that in most cases, the initial representation is intractable.

To illustrate the basic principles behind our technique, we apply them to a small worked example. The example is described in the next section.

5 The Simple System in Promela

The example is taken from figure 2, with two processes User 0 and User 1, instantiations of a generic process given in Promela by the following:

```
mtype = { send, receive, u0, u1 };
mtype lastaction[2];
proctype user(chan inchan, outchan;mtype myid, oppid)
{
        do
        ::outchan!(myid,oppid) ->
                lastaction[myid] = send
        ::inchan?eval(oppid),eval(myid) ->
                lastaction[myid] = receive
        od
}
```

Promela has a C-like syntax. Briefly, `mtype` is a Promela keyword for an enumerated type; so `send`, `receive`, etc. are constants. `c!m` denotes write `m` on channel `c`, `c?m` denotes read `m` from channel `c` (destructive read). Statements (e.g. assignments denoted by "=") can be guarded by other statements, with the form statement `->` statement. The second statement executes only if the guard is not blocked. The eval functions ensure that incoming messages must match these variable values rather than overwriting them.

All channels are parameters: this allows us to instantiate the process with either communications structures from figure 2 without having to alter the body of the process. Promela representations to be used with our method must be encoded in this way to allow the communication structure to be altered without other changes to the representation.

The whole system consists of two instantiations of the process `user`, thus assuming that `chans` is an array of channels, the initial representation would use a separate channel for each parameter as shown below:

```
        run user(chans[0], chans[1], u0, u1);
        run user(chans[1], chans[0], u1, u0)
```

and the candidate representation would use the same channel thus:

```
run user(chans[0], chans[0], u0, u1);
run user(chans[0], chans[0], u1, u0).
```

Note also that the send and receive statements in the process definition include variables denoting the intended recipient of the message. This is to prevent a user process receiving its own message in a shared channel configuration. This annotation of messages is necessary to ensure messages arrive as intended regardless of communication structure.

5.1 Specified Properties

In Spin, properties are expressed using linear temporal logic (LTL) [11]. The properties are as follows:

– *User0 will eventually receive a message.* This is expressed in LTL as $\Box\Diamond p$ where p is the boolean expression lastaction[0]==receive. We refer to this property as **user0 receive**.
– *There exists a reachable state where the last message action for both User0 and User1 are receive actions.* This is expressed in LTL as $\Diamond(p \wedge q)$ where p is the boolean expression lastaction[0]==receive and q is the boolean expression lastaction[1]==receive. We refer to this property as **both receive**.

We now return to the task of testing for property preservation.

6 Property Preservation Testing

The communication structure alteration procedure described in section 4 requires that we determine whether a specified property is preserved between two alternative communication structures. In this section we present an overview of our procedure for testing for property preservation which involves comparing traces in *message automata.*

6.1 Message Automata

To compare alternative representations we use *message automata*, an abstraction we have devised to reason about communication structure. The message automata for example system are shown in figure 3, the initial model on the left and the candidate model is on the right.

Message automata consist of *message states* linked by *message statements.* A message state is labelled with a name and the messages that are present on all channels, messages that have been sent but not yet received, at that state. We are only interested in whether messages have been sent or received: which channels are used for their transit is irrelevant. A message statement is a Promela statement that sends or receives a message, here prefixed by its (local) process name.

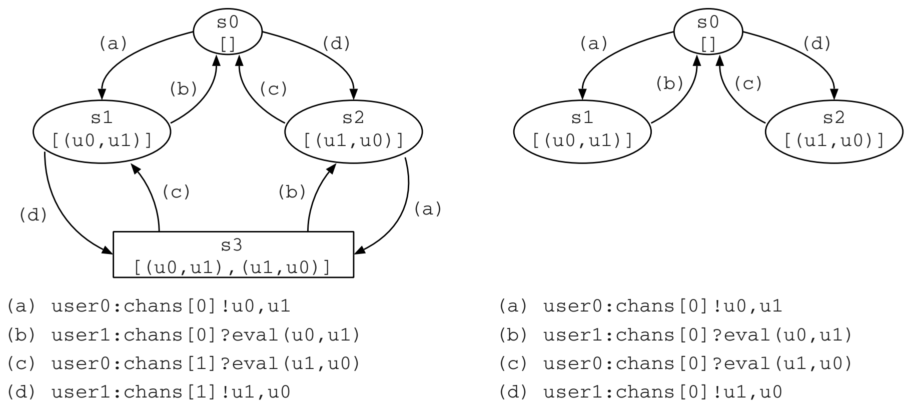
The two message automata in figure 3 differ by only a single state: the state s3. This non-shared state is known as a *difference state* (denoted by a rectangle) and is crucial to determining whether properties are preserved between the two communication structures.

6.2 Traces

The key idea behind testing for property preservation is comparing *traces* through the message automaton. In particular, we are concerned with *difference traces* and *emulating traces*. A difference trace is a trace which exists in the message automaton for one communication structure, but not for the other. In the simple example, the trace s0s1s3 is a difference trace, since it can only be achieved by the message automaton for the initial system. By definition, every difference trace contains at least one difference state. An emulating trace is a trace which *emulates* the behaviour of a difference trace with respect to a specified property. To emulate a difference trace, the emulating trace must match both the initial and final message states and must also match the *effect* of the trace on a specified property. We will describe the exact meaning of *effect* in section 6.4, in the next section we discuss how to reduce the number of traces under consideration.

6.3 Difference Sub-traces

To reduce the emulation effort, we will emulate only difference *sub*-traces, illustrated in figure 4. The figure shows two disjoint sets of states (think of them as rings, this is not a Venn diagram): an inner set containing the states shared by the two message automata and an outer set containing the difference states. In our example, the shared set would contain s0, s1 and s2 and the difference set s3.



```
(a) user0:chans[0]!u0,u1
(b) user1:chans[0]?eval(u0,u1)
(c) user0:chans[1]?eval(u1,u0)
(d) user1:chans[1]!u1,u0
```

```
(a) user0:chans[0]!u0,u1
(b) user1:chans[0]?eval(u0,u1)
(c) user0:chans[0]?eval(u1,u0)
(d) user1:chans[0]!u1,u0
```

Fig. 3. Message automata for the simple system. (left) Initial communication structure. (right) Candidate communication structure.

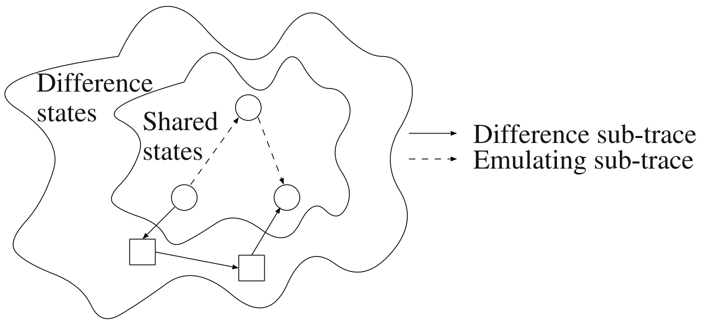


Fig. 4. Emulating sub-trace illustration.

When emulating a trace, any sub-trace which exists only within the shared set can be emulated by simply copying the appropriate transitions: the states are common to both automata. Only when a trace enters the difference set is more sophisticated emulation required. We can take advantage of this observation by only emulating the difference sub-traces, the sections of a trace which enter the difference set. Once the trace re-enters the shared set, direct emulation is possible: an emulating trace for this section already exists since the shared set of states and their transitions is identical in the two message automata.

From the simple example, consider the trace s0s1s3s2s0. The subtraces s0s1 and s2s0 use only shared states and can be emulated directly. We need only find an emulating sub-trace for the difference sub-trace s1s3s2. This emulating sub-trace must not only match the effect on a specified property, but also the start and end states of the difference sub-trace. This would allow the emulating sub-trace to form a direct substitution for the difference sub-trace as part of a longer trace. In the example above, assume s1s0s2 is an emulating sub-trace for the difference sub-trace s1s3s2. We can now use the emulating sub-trace to substitute in the complete trace. The trace s0s1s0s2s0 has the same effect on a specified property and contains no difference states: it is an emulating trace.

This principle can also be applied to traces of infinite length. For example, consider the difference trace s0s1s3s2s0 where these states cycle infinitely often. Assume that s1s0s2 is an emulating sub-trace for the difference sub-trace s1s3s2. On each occasion the difference trace enters the difference sub-trace s1s3s2 we can substitute the emulating trace s1s0s2; this provides an emulation of the infinite trace.

Our approach is therefore to identify every possible difference sub-trace and then to find an *emulating sub-trace* to emulate its behaviour using only non-difference states.

This method can be thought of as an exhaustive search, as would be performed by a Spin verification, of the difference behaviour for a system. As with Spin, we are able to cope with traces that are potentially infinite by capturing the finite number of effects on a property these traces may cause. With a finite number of effects, and a finite number of states which could begin and end the difference sub-traces, there are also a finite number of difference sub-traces. Some exceptional cases will be discussed in section 6.6.

Note that an emulating sub-trace need not contain the same number of states as its difference sub-trace, provided it starts and ends in the same place and emulates the behaviour with respect to a specified property. How we determine this behaviour is discussed in the next section.

6.4 Trace Effect

In Spin, a property is represented as a Büchi automaton. Transitions between states are labelled by propositional logic formulae where the propositions are boolean expressions from the Promela model, for example, the propositions p and q from the examples in section 5.1. To verify a property, the automaton is treated as a process and run concurrently (synchronously) with the model, with property transitions traversed as the labelling conditions become true. It is the sequence of property states which dictates whether the property will be true or false. For more on the theoretical background to model checking temporal properties, see [4].

To determine the effect of a trace, such as a difference trace, we determine what variables will have values assigned by the statements associated with a trace. From this, we can identify what possible sequences of property states will occur. To ensure enough information for this analysis, we must carry in each trace the possible property states along with the relevant variable values as they are altered. By emulating difference sub-traces we make no assumption about the property states or variable values at the start of the sub-trace and therefore must check all combinations.

Consider the example difference sub-trace s1s3s2 and the specified property **user0 receive**. The difference trace s1s3s2 corresponds to the message statements {user1:chans[1]!u1,u0; user1:chans[0]?eval(u0,u1)}. By examining the Promela representation in section 5 we can see that the first message event causes the assignment lastaction[myid] = send. This assignment alters the value of the variable lastaction[0] (for this particular process instance), which is referenced in the proposition p in the property **user0 receive**. In this case, the value is set to send, making the proposition false. This effect on the truth value of the proposition will cause some transitions in the property automaton: it is this effect that we must emulate. Note that other statements may only alter variables which have no effect upon the specified property: such statements can be safely ignored.

6.5 Emulation Checking

So far we have discussed only emulating an individual difference trace, but property preservation testing involves checking *all* difference sub-traces. We call this procedure *emulation checking*. Emulation checking works in two stages:

1. Identify all difference sub-traces and their effect on the specified property.
2. Attempt to find an emulating sub-trace for each difference sub-trace.

In the first step, to identify all difference sub-traces, we identify all the message states at which there is a transition to a difference state. From these states we find all the traces that contain a number of difference states concluding with a non-difference state. In the simple example, from the state s1 there are difference sub-traces s1s3s1 and s1s3s2. From the state s2 there are difference sub-traces s2s3s2 and s3s2s1. The effect of each difference sub-trace, on the specified properties, is then determined by the method described in section 6.4.

In the second step, we attempt to find emulating sub-traces for each difference sub-trace. This part of the procedure is uncertain because the identification of emulating sub-traces is based on heuristic search. There may be a variety of routes that match the start and end states of a difference sub-trace and pass through only non-difference states but we must find one which also matches the effect of the difference sub-trace. Details of how we carry out this search and various optimisations are detailed in [13].

6.6 Further Detail

Due to space constraints we have not described the full detail of the property preservation test here. This includes dealing with ambiguous message states, addressing infinite difference traces and reducing the effort of calculating trace effect. This detail can be found [13].

7 Soundness and Completeness

It is crucial that our method is sound. If it is not sound, a user could believe that a property was preserved when in fact it is not. This could lead to an unsafe change to a communication structure — obviously undesirable. If our method is not complete, some safe alteration will be rejected as not property preserving, but then the user is back to where they started. Although they cannot verify their system, at least they do not have an untrustworthy verification result. Our method aims to be sound, but not necessarily complete. (Note, an algorithmic method cannot be complete, if the initial representation is intractable.)

Throughout section 6 we have implicitly justified the soundness of our technique. In section 6.3 we described how even infinite difference traces must consist of parts which can be emulated directly and difference sub-traces, of which there are a finite number. In section 6.4 we described how to ensure we capture all the possible behaviours which may be due to a difference sub-trace. In fact, there are some cases in which our method is not sound, based on the use of *control variables* within the Promela representation. These are relatively specialised cases and can be easily identified; for more detail, see [13].

The method is not complete because, as discussed in section 6.5, identifying all emulating traces can be a difficult task which may not be achieved. In fact, even if the identification of emulating traces was perfect, our method would still not be complete because the method only determines whether any unique behaviour exists, not whether this behaviour necessarily alters the verification of a specified property.

8 Implementation

We have implemented a tool based on the techniques described in section 6. The tool is approximately 8000 lines written in the scripting language Python [8] and interfaces with Spin as well as providing output via the graph drawing package dot [7]. As input, the tool requires a system modelled in Promela using two different communication structures and a specified property in the file format used by Spin. As output, it returns whether the property is preserved between the two structures. If the property is not preserved, it provides a trace for one communication structure which cannot be emulated by the other. This tool was used to generate the results presented in section 10.

9 Communication Structure Selection

Whether we are constructing a new representation or applying communication structure alteration, we must be able to choose a communication structure that will result in a small(er) state space. To achieve this, we have devised a set of guidelines for choosing a communication structure.

In general, we cannot know which communication structure will result in the smallest state space without model checking all conceivable communication structures. So, for some unusual systems, these guidelines may not provide the best choice of state space. However, intuition and empirical observation suggest that they are effective in most cases.

Few Channels. As demonstrated by the example in section 2, a smaller number of channels results in fewer combinations of messages. This in turn should result in a smaller state space.

Short Channels. As with few channels, the shorter the channels the fewer combinations and therefore the smaller the state space.

Use Exclusive Send/Exclusive Receive. Spin includes constructs to improve the application of partial order reduction [10] to communication operations. As far as possible within the other guidelines, channel structures which make use of these constructs should be employed.

Avoid Too Many Shared Channels. If channels are shared between more than 2 groups of processes it is very common for deadlocks to be created. To maintain a model without deadlocks, too many shared channels should be avoided.

10 Case Study

To illustrate our method, we expand the simple example from section 2.

10.1 System Description

The system we have chosen to model is based on the simple system with three users, where each process is an Email relay server, transmitting Email messages to the other servers. Once the Email arrives at the server it is either read or discarded by its intended recipient. To introduce some extra complexity into the model, each relay is capable of sending either legitimate Email or unsolicited junk Email, also known as *spam*. If a particular relay sends more than a fixed number of spam Emails, it is placed on a *block list*, and no further Emails from this relay are read — they are received, but immediately discarded. The identification of a message as spam is assumed to be perfect.

10.2 Specified Properties

We consider three specified properties, which we describe informally by:

– **no spam** spam messages are received but are not read.
– **blocked** once a relay is blocked, its messages are never read again.
– **arrival** unless a relay is blocked, sent non-spam messages are eventually read.

10.3 Communication Structures

The communication structures are as shown in figure 2 with three users. The initial structure, i.e. case (a), uses dedicated channels, one for each connection. The candidate structure, i.e. case (b), uses shared channels, where one channel is used to connect each pair of processes. In both cases all channels are of size 1.

10.4 Property Preservation Test Results

Applying our property preservation testing method to the example shows that all three properties are preserved. For detailed statistics about each run, see [13]. In summary, in the worst case the message automata for the initial and candidate representations were of size 1745 and 112 respectively and this resulted in a total of 356 difference sub-traces comprising 11970 trace states. The longest run of the tool took around half an hour.

10.5 Spin Results

Table 2 shows the results of using Spin to verify the specified properties with both representations[1]. In this table, N/A indicates that Spin was unable to complete an exhaustive search with the available memory.

From this table we can see that in each case, the number of states in the candidate communication structure is less than that of the initial communication structure. In the case of the properties **blocked** and **arrival**, state spaces which were previously intractable — too large to fit into available memory — have been made tractable by altering the communication structure. In the **no spam** case, where both the initial and the candidate models are tractable, Spin confirms that the properties are preserved (and the candidate state space is smaller).

This case study shows that our method yields smaller state spaces, and is applicable to systems with (initially) intractable state spaces.

[1] These tests were run with a PC running Linux, with memory set to 1GB.

Table 2. Spin verification results.

	Result		States Stored	
	Initial	Candidate	Initial	Candidate
no spam	True	True	6.67341e+06	1.27515e+06
blocked	?	True	N/A	5.59176e+06
arrival	?	True	N/A	9.11464e+06

11 Related Work

The idea that alteration of a Promela representation to reduce the state space has also been presented in [12]. The results presented in [12] is more from the perspective of a Promela programmers guide, whereas we aim for a more rigorous semantic equivalence.

The construction of a communication automaton is similar to the use of *slicing* [14,9], since it slices away non-communication behaviour. Our method differs from slicing in that it attempts a more radical alteration of the Promela code. Instead of simply trimming away parts of the model which are unnecessary, we alter the underlying structure of the system.

In [3], the authors describe the abstraction of communication channels in Promela to explore an otherwise intractable search space. They advocate reducing the number of messages that are passed. Like our work, [3] recognises the significance of communication however, the authors apply abstraction to the *messages* on the channels, rather than by altering the communication structure.

12 Limitations and Further Work

The difficulty of identifying emulating traces makes our prototype tool fairly slow: some of the results presented took several hours to generate. A more efficient implementation would make the method more accessible.

The communication structure choice guidelines are a good starting point, but would be difficult to apply them automatically. An algorithmic method for choosing good communication structures would be preferable.

13 Conclusion

We have proposed a method for reducing the state space size for a model of a communicating system by altering the communication structure. We have provided guidelines for choosing an appropriate communication structure, and presented a method for determining property preservation between two models with different communication structures. We have also presented an example which shows these principles in action. The major contribution is a method which is applicable even when the initial representation is intractable. The method is sound (except for a few special cases), but not necessarily complete.

References

1. Muffy Calder and Alice Miller. Using SPIN for feature interaction analysis - a case study. In *Lecture Notes in Computer Science*, volume 2057 of *Proceedings of the 8th International SPIN Workshop (SPIN2001)*, pages 143–162, May 2001.
2. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
3. E. Fersman and B. Jonsson. Abstraction of communication channels in promela: A case study, 2000.
4. Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.
5. Gerard Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
6. G.J. Holzmann and Doron Peled. An improvement in formal verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211, Berne, Switzerland, October 1994. Chapman & Hall.
7. Eleftherios Koutsofios and Stephen North. Drawing graphs with *dot*. Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ, USA, September 1991.
8. Mark Lutz. *Programming Python*. O'Reilly & Associates, Inc., 103a Morris Street, Sebastopol, CA 95472, USA, Tel: +1 707 829 0515, and 90 Sherman Street, Cambridge, MA 02140, USA, Tel: +1 617 354 5800, Fall 1996.
9. L. Millett and T. Teitelbaum. Slicing promela and its applications to model checking, 1998.
10. D. Peled. Combining partial order reductions with on-the-fly model-checking. *Lecture Notes in Computer Science*, 818:377–390, 1994.
11. Doron Peled. Partial order reduction: Linear and branching temporal logics and process algebras. In *Partial Orders Methods in Verification*, DIMACS, pages 233–257, Princeton, NJ, USA, 1996. American Mathematical Society.
12. Theo C. Ruys. Low-fat recipes for spin. In *Lecture Notes in Computer Science*, volume 1885 of *7th SPIN workshop*, pages 287–321. Springer Verlag, sep 2000.
13. Peter Saffrey. *Optimising Communication Structure for Model Checking*. PhD thesis, Department of Computing Science, University of Glasgow, July 2003.
14. Frank Tip. A survey of program slicing techniques. Technical Report CS-R9438, CWI - Centrum voor Wiskunde en Informatica, July 31, 1994.

Translating Software Designs for Model Checking[*]

Fei Xie[1], Vladimir Levin[2], Robert P. Kurshan[3], and James C. Browne[1]

[1] Dept. of Computer Sciences, Univ. of Texas at Austin, Austin, TX 78712
{feixie,browne}@cs.utexas.edu
Fax: +1 (512) 471-8885

[2] Microsoft, One Microsoft Way, Redmond, WA 98052
vladlev@microsoft.com
Fax: +1 (425) 936-7329

[3] Cadence, 35 Spring St., New Providence, NJ 07974
rkurshan@cadence.com
Fax: +1 (908) 898-1435

Abstract. This paper presents a systematic consideration of the major issues involved in translation of executable design level software specification languages to directly model-checkable formal languages. These issues are considered under the framework of *integrated model/property translation* and include: (1) translator architecture; (2) semantics translation from a software language to a formal language; (3) property specification and translation; (4) transformations for state space reduction; (5) translator validation and evolution. Solutions to these issues are defined, described, and illustrated in the context of translating xUML, an executable design level software specification language, to S/R, the input formal language of the COSPAN model checker.

1 Introduction and Overview

Model checking [1,2] has major potential for improving reliability of software systems. Approaches to software model checking can be roughly categorized as follows:

1. Manually creating a model of a software system in a directly model-checkable formal language and model checking the model in lieu of the system;
2. Subsetting a software implementation language and directly model checking programs written in this subset;
3. Subsetting a software implementation language and translating this subset to a directly model-checkable formal language;
4. Abstracting a system implemented in a software implementation language and translating the abstraction into a directly model-checkable formal language;
5. Developing a system in an executable design level software specification language and translating the design into a directly model-checkable formal language;
6. Model checking a property on a system through systematic testing of the execution paths associated with the property.

[*] This research was partially supported by NSF grant 010-3725.

Categories 3, 4, and 5 cover a large fraction of the approaches to software model checking, such as [3,4,5,6,7,8], all of which require translation from a software language or an abstraction specification language to a directly model-checkable formal language. Translation helps avoid the "many models" problem: as a system evolves, models of the system are manually created and may contain errors or inconsistencies. Translation also enables application of state space reduction algorithms by transforming the designs, implementations, and abstractions being translated. There has, however, been little systematic consideration of issues involved in translating software specification languages used in software development to directly model-checkable formal languages.

This paper identifies and formulates several major issues in translating executable design level software specification languages to directly model-checkable formal languages. Solutions to these issues are defined, described, and illustrated in the context of developing the translator [8] from xUML [9], an executable design level specification language, to S/R [10], the input language of the COSPAN [10] model checker. (Another translator [11], which translates SDL [12] to S/R, is also referred to as we discuss issues related to reuse of translator implementation.)

Model checking of a property on a software system via translation *only* requires that the behaviors of the system related to the property be preserved in the resulting formal model. The artifact to be translated consists of a model of a software system and a property to be checked. This *integrated model/property translation* provides a natural framework for generating a formal model that preserves only the behaviors required for model checking a given property and has a minimal state space. Under this framework, the following issues in translation of executable design level software specification languages have been identified and formulated in developing the xUML-to-S/R translator:

– **Translator architecture.** The architecture of translators should simplify implementation and validation of translation algorithms and transformation algorithms for state space reduction, and also enable reuse of these algorithms.
– **Semantics translation from a software language to a formal language.** Model checking of software through translation requires correct semantics translation from a software specification language to its target formal language. The semantics of the source software language and the semantics of the target formal language may differ significantly, which may make the translation non-trivial.
– **Property specification and translation.** Effective model checking of software requires specification of properties on the software level and also requires integrated translation of these properties into formal languages with the system to be checked.
– **Transformations for state space reduction.** Many state space reduction algorithms can be implemented as source-to-source transformations in translators.
– **Translator validation and evolution.** Translators must be validated for correctness. They must be able to adapt to evolution of source software languages and target formal languages, and incorporation of new state space reduction algorithms.

These issues arise generally in translation of software specification languages for model checking. We have chosen executable design level software specification languages as our representations for software systems for the following reasons:

- These languages are becoming increasingly popular in industry and development environments for these languages are commercially available.
- These languages have complete execution semantics that enable application of testing for validation and also enable application of model checking for verification.
- A design in these languages can be compiled into implementation level software specification languages and also can be translated into directly model-checkable formal languages. This establishes a mapping between the implementation of the design and the formal model of the design that is model checked, which avoids the "many models" problem.
- These languages require minimal subsetting to enable translation to directly model-checkable formal languages.

The balance of this paper is organized as follows. In Sections 2, 3, 4, 5, and 6, we elaborate on these issues and discuss their solutions in the context of the xUML-to-S/R translator. We summarize several case studies using the xUML-to-S/R translator in Section 7, discuss related work in Section 8, and conclude in Section 9.

2 Translator Architecture

This section presents a general architecture for translators from software specification languages to directly model-checkable formal languages and briefly discusses the functionality of each component in this architecture. The emphasis is on the Common Abstraction Representation (CAR), the intermediate representation of the translation process. Many of the important functionalities of translators are implemented as source-to-source transformations on the software model to be translated or on the CAR.

2.1 A General Architecture for Translators

A general architecture for translators is shown in Figure 1. A notable feature of this architecture is that the software model and the property to be checked on the model are processed in an integrated fashion by each component. The frontend of the translator not only constructs the Abstract Syntax Tree (AST) of the software model, but also transforms the AST with respect to the property by applying source-to-source transformations such as the loop abstraction [13]. These transformations are partially guided by directives written in an annotation language to be discussed in Section 5.1. Functionalities of other components are discussed in Section 2.2 after we introduce the CAR.

2.2 Common Abstraction Representation (CAR)

A CAR is a common intermediate representation for translating several different software languages. It captures abstract concepts of the basic semantic entities of these languages and is designed to be a minimal representation of the core semantics of these languages. A CAR has been derived for the development of the xUML-to-S/R and SDL-to-S/R translations. The basic entities in this CAR include a system, a process, a process buffer, a message type, a message, a variable type, a variable, and an action. A process entity is

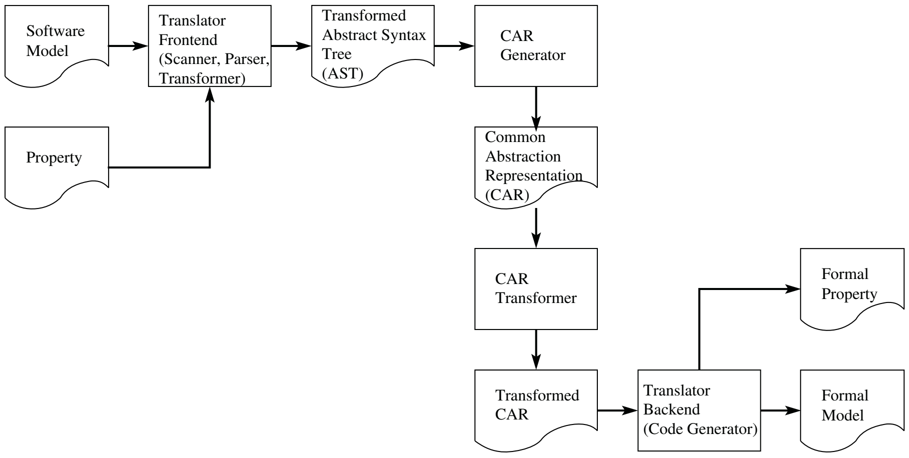

Software Model
Translator Frontend (Scanner, Parser, Transformer)
Transformed Abstract Syntax Tree (AST)
CAR Generator
Property
Common Abstraction Representation (CAR)
CAR Transformer
Formal Property
Transformed CAR
Translator Backend (Code Generator)
Formal Model

Fig. 1. Translator architecture

structured as a graph whose nodes are states, conditions, and actions and whose edges are transitions. Actions are input, output, assignment, and etc.

Entities in a CAR may have parameterized definitions. Semantics of such entities can be exactly specified only by referring to a specific source language. For instance, in an xUML process, actions are associated with states while in a SDL process, actions are associated with transitions. For translation of a specific source language, a *profile* of the CAR is defined. The profile is a realization of the CAR which includes the CAR entities necessary for representing the source language and realizes the CAR entities with parameterized definitions according to the semantics of the source language. Each model in the source language is represented by an instance of the CAR profile. The CAR profile thus inherits its semantics from the source language. This semantics is mapped to the semantics of a target language by a translator backend. CAR profiles for different source languages require different translation backends to a target language. These backends share translation procedures for a CAR entity if the entity has the same semantics in the corresponding source languages. Semantic entities of a source language that are not in the CAR are either reduced to the entities that are in the CAR or included as extensions in the CAR profile for the source language. Having a CAR and different CAR profiles for different source languages offers the following benefits:

– A CAR profile only contains the necessary semantic entities for a source language. Therefore, it is easier to construct and validate the translation from the CAR profile to the target language than the direct source-to-target translation.
– The simplicity of the CAR profile simplifies the implementation and validation of transformations for state space reduction.
– The CAR enables reuse of the translation algorithms and the transformation algorithms for the semantic entities shared by different profiles of the CAR.

There is often a significant semantics gap between a source language and a target language, which makes a single-phase direct translation difficult. Having a CAR allows us to divide the translation from a source language to a target language into three phases: (1) the *CAR instance construction* phase, (2) the *CAR instance transformation* phase, and (3) the *target language code generation* phase.

In the *CAR instance construction* phase, a model in the source language is scanned, parsed, and transformed, and a CAR instance is then constructed. Complex semantic entities in the model are reduced to basic semantic entities in the CAR. For instance, in xUML, there are several different loop structures in the action language such as a *for* loop, a *while* loop, and a *do* loop. All these loop structures are reduced to a simple loop structure composed of a condition, the loop body, and *goto* actions. Implicit semantic entities are made explicit in the CAR instance. For instance, there is an implicit message buffer for each class instance in an xUML model, which is not explicitly represented in the xUML model. To be translated, such buffers are made explicit.

In the *CAR instance transformation* phase, the CAR instance is transformed by source-to-source transformations for state space reduction. CAR provides a common representation on which transformations for state space reduction such as static partial order reduction [14] can be implemented. Since the CAR profiles for different source languages share semantic entities, transformations implemented on these semantic entities may be reused in translation of different source languages.

In the *target language code generation* phase, a model in the target language is generated from the transformed CAR instance. For each semantic entity in the CAR, a code generation procedure is defined. As the AST of a CAR instance is traversed, if a semantic entity is identified, the corresponding code generation procedure is invoked to emit codes in the target language. An entity in the CAR may have different semantics when used in translation of different source languages. Therefore, the code generation procedures for translating this entity may be different for different source languages. For instance, in xUML each class instance has a message buffer while in SDL each process has a message buffer. However, in xUML and SDL the message buffers have different semantics. In xUML, a class instance can consume, discard, and throw an exception on a message in its message buffer. In SDL, a process can save a message in its buffer and consume it in the future. The translation procedures for translating an xUML message buffer and an SDL message buffer are, therefore, different.

3 Semantics Translation from Software Language to Formal Language

To translate a software language for model checking, a proper target formal language must be selected. After selection of the target language, a translatable subset of the software language is derived. This subset is mapped to the CAR by reducing complex semantic entities in the source language to simple semantic entities in the CAR. The simplified semantics of the source language is then simulated with the semantics of the target language. We discuss these steps in the context of the xUML-to-S/R translation.

3.1 Selecting Target Formal Language

There are many directly model-checkable formal languages. Promela [15], SMV [16], and S/R [10] are among the most widely used. These languages have various semantics.

Their corresponding model checkers, SPIN [15], SMV [16], and COSPAN [10], support different sets of search algorithms and state space reduction algorithms. Appropriate selection of a target formal language should consider three factors: *application domain*, *semantics similarity*, and *model checker support*. These three factors were considered synergistically in selecting the target language for translating xUML.

– **Application domain.** While this paper is concerned only with translation of a software design, the ultimate goal of this project is hardware/software co-verification. xUML is widely used in development of embedded systems which often requires hardware/software co-design and co-verification. Such a system, at least on different levels of abstraction, may exhibit both hardware-specific (tighter synchronization) and software-specific (looser synchronization) behaviors.
– **Semantics similarity.** The asynchronous interleaving semantics of xUML is close to the semantics of Promela, which would simplify the translation, while both SMV and S/R have synchronous parallel execution semantics.
– **Model checker support.** In practical model checking, especially in co-verification, the widest range of search algorithms and state space reduction algorithms is desired since it is not clear that any of these algorithms is superior for a well-identified class of systems. A system that has both software and hardware components may often benefit from symbolic search algorithms based on BDDs and SAT solvers which are not available in SPIN. SMV provides BDDs and SAT based symbolic search algorithms. However, Depth-First Search (DFS) algorithms with explicit state enumeration, which have demonstrated their effectiveness in verification of many software-intensive systems, are not available in SMV. COSPAN offers both symbolic search algorithms and DFSs with explicit state enumeration. In addition, COSPAN supports a wide range of state space reduction algorithms such as localization reduction [10], static partial order reduction [14], and a prototype implementation [17] of predicate abstraction.

Based on the above, we selected S/R as the target language at the cost of a non-trivial xUML-to-S/R translation.

3.2 Subsetting Software Language

Software languages such as xUML may have multiple operational semantics and may also have semantic entities not directly translatable to the selected target language. For model checking purposes, a subset of the software language must be derived for a given application domain. This subset must have a clean operational semantics suitable for the application domain. Semantic entities that are not directly translatable, such as continuous data types, must be either excluded from the subset or discretized and simulated by other semantic entities. Infinite-state semantic entities may be directly translated or be bounded and then translated depending on whether the target language supports infinite-state semantic entities or not. If a target formal language permits some infinite-state semantic entities, necessary annotations may also need to be introduced for the subset so that infinite-state semantic entities in the subset can be properly translated.

In the xUML-to-S/R translation, we adopt an asynchronous interleaving semantics of xUML (see Section 3.4) while xUML has other semantics such as asynchronous parallel. Continuous data types such as float can be simulated by discrete data types such as integer if such a simulation does not affect the model checking result. Since S/R does not support infinite semantic entities, infinite data types and infinite message queues must be bounded implicitly by convention or explicitly by user annotations.

3.3 Mapping Source Software Language to CAR

After the translatable subset of the source software language is derived, a CAR profile is identified accordingly. The CAR profile only contains the basic entities necessary for representing the source language subset. A mapping is then established from the source language subset to the CAR profile. Complex semantic entities in the source language are reduced to simple semantic entities in the CAR. For instance, in xUML a state action can be a collection action that applies a sub-action to elements of a collection in sequence. The collection action is reduced into a loop action with a test checking whether there still are untouched elements in the collection, and with the sub-action as the loop body. After the mapping is established, the semantics of the CAR profile is decided by the semantics of the source language and the mapping.

3.4 Simulating Source Semantics with Target Semantics

The mapping from the source language to the CAR profile removes complex semantic entities from the source semantics. To complete the translation from the source language to the target language, only this simplified form of the source semantics must be simulated with the target semantics. We first sketch the semantics of xUML and S/R, then discuss how the asynchronous semantics of xUML is simulated with the synchronous semantics of S/R and how the run-to-completion requirement of xUML is simulated.

Background: Semantics of xUML and S/R. xUML has an asynchronous interleaving message-passing semantics. In xUML, a system consists of a set of class instances. Class instances communicate via asynchronous message-passing. The behavior of each class instance is specified by an extended Moore state model in which each state may be associated with a state action. A state action is a program segment that executes upon entry to the state. In an execution of the system, at any given moment only one class instance progresses by executing a state transition or a state action in its extended Moore state model. S/R has a synchronous parallel semantics. In S/R, a system consists of a set of automata. Automata communicate synchronously by exporting variables to other automata and importing variables from other automata. The system progresses according to a logical clock. In each logical clock cycle, each automaton moves to its next state according to its current state and the values of the variables it imports.

Simulation of Asynchrony with Synchrony. The asynchronous interleaving execution of an xUML system is simulated by the synchronous parallel execution of its corresponding S/R system as follows. Each class instance in the xUML system is mapped an automaton in the S/R system. An additional automaton, *scheduler*, is introduced in the S/R system. The *scheduler* exports a variable, *selection*, which is imported by each S/R automaton corresponding to an xUML class instance. At any given moment, the *scheduler* selects one of such automata through setting *selection* to a particular value.

Only the selected automaton executes a state transition corresponding to a state transition or a state action in the corresponding xUML class instance. Other automata follow a self-loop state transition back to their current states.

The asynchronous message-passing of xUML is simulated by synchronous variable-sharing of S/R through modeling the message queue of a class instance as a separate S/R automaton. Let automata IP_1 and IP_2 model two class instances and automata QP_1 and QP_2 model their corresponding private message queues. The asynchronous passing of a message, m, from IP_1 to IP_2 is simulated as follows: [1: $IP_1 \to QP_2$] IP_1 passes m to QP_2 through synchronous communication; [2: Buffered] QP_2 keeps m until IP_2 is ready for consuming a message and m is the first message in the queue modeled by QP_2. [3: $QP_2 \to IP_2$] QP_2 passes m to IP_2 through synchronous communication.

Simulation of Run-to-Completion Execution. A semantic requirement of xUML is the run-to-completion execution of state actions, i.e., the executable statements in a state action must be executed consecutively without being interleaved with state transitions or executable statements from other state actions. This run-to-completion requirement is simulated as follows. An additional variable, *in-action*, is added to each S/R automaton corresponding to an xUML class instance. All *in-action* variables are imported by the *scheduler*. When an automaton is scheduled to execute the first statement in a state action, it sets its *in-action* to true. When the automaton has completed with the last statement in the state action, it sets its *in-action* to false. The scheduler continuously schedules the automaton until its *in-action* is set to false.

4 Property Specification and Translation

Since the entire translation process is property-dependent, properties must be specified at the level of and in the name space of software systems. Additionally, software level property specification enables software engineers who are not experts in model checking to formulate properties. We discuss software level property specification and translation of software level properties in terms of xUML and a linear-time property specification language, but the arguments carry over for other software specifications and temporal logics. Two issues related to property specification and translation: (1) automatic generation of properties and (2) translation support for compositional reasoning, conclude this section.

4.1 Software Level Property Specification

An xUML level property specification language, which is linear-time and with the expressiveness of ω-automata, has been defined. This language consists of a set of property templates that have intuitive meanings and also rigorous mappings into the FormalCheck property specification language [18] which is written in S/R. The templates define parameterized automata. Additional templates can be formulated in terms of the given ones, if doing so simplifies the property specification process. A property formulated in this language consists of declarations of propositional logic predicates over semantic entities of an xUML model and declarations of temporal predicates. A temporal predicate is declared by instantiating a property specification template: each argument of the template is replaced by a propositional logic expression composed from previously declared propositional predicates.

To further simplify property specification, for an application domain, frequently used property templates and customized property templates are included in a domain-specific property template library based on previous verification studies in the domain. These property templates are associated with domain-specific knowledge to help software engineers select the appropriate property templates. A similar pattern-based approach to property specification was proposed by Dwyer, Avrunin, and Corbett in [19].

4.2 Property Translation

To support the integrated model/property translation, once the property specification language is defined, semantic entities for representing properties are introduced as extensions to the CAR profile for the source software language. A model and a property to be checked on the model are integrated in an instance of the CAR profile. In the xUML-to-S/R translation, properties are translated by a module of the translator. Since a property refers to semantic entities in the xUML model to be checked, this module conducts syntax and semantic checking on a property by referring to the abstract syntax tree constructed from the model. For each property template, a translation procedure is provided, which maps an instance of the template to the corresponding semantic entity in the CAR profile and ultimately to a property in S/R for use by COSPAN.

4.3 Automatic Generation of Properties

Certain types of properties, such as safety properties that check buffer overflows, can be automatically generated during translation. Translators can apply static analysis techniques that identify implicit buffers and generate properties for checking possible overflows of these buffers. For instance, in xUML, every class instance has an implicit message buffer, which has the risk of buffer overflow. The xUML-to-S/R translator automatically generates a safety property for each message buffer. When the resulting S/R model is model checked, the safety property will catch any buffer overflow related to the message buffer being monitored. Automatically generated properties are integrated into translation in the same way as user-defined properties.

4.4 Translation Support for Compositional Reasoning

Another application of the software level property specification language is in constructing abstractions of components to be used in compositional reasoning [20] where model checking a property on a system is accomplished by decomposing the system into components, checking component properties locally on the components, and deriving the property of the system from the component properties. A property of a component is model-checked on the component by assuming that a set of properties hold on other components in the system. These assumed properties are abstractions of other components in the system and are used to create the closed system on which the property of the target component to be verified is model checked. These properties are formulated in the software level property specification language. The assumed properties on other components are called the *environment assumptions* of the target component. To support compositional reasoning, the translator is required to support translation of a closed system that consists of a component of a system and the environment assumptions of the component. This is in contrast to model checking without compositional reasoning where the translator is only required to support translation of a closed system that consists purely of entities specified in the software language, xUML in our case.

5 Transformations for State Space Reduction

The ultimate goal of integrated model/property translation is to generate a formal model which preserves only the behaviors of the source software model required for model checking a specific property and which has a minimal state space. Many state space reduction algorithms can be implemented as source-to-source transformations in the translation. This section describes model transformations implemented in the xUML-to-S/R translation and a model annotation language used to specify some types of transformations. Similar transformations will surely be applied in translation from most software specification languages to directly model-checkable formal languages.

5.1 Model Annotation Languages

There is often domain-specific information that is not available in a software model, but can facilitate transformations for state space reduction, for instance, bounds for variables in the software model. Software engineers can introduce such information by annotating the model with an annotation language before the model is translated. Such annotations are introduced in an xUML model as comments with special structures so that they will not affect other tools for xUML, for instance, xUML model execution simulators. The annotations must be updated accordingly as the model is updated.

Variable bounds are introduced in an xUML model as annotations associated with the variables or the data types of the variables. Annotation-based variable bounding indirectly enables symbolic model checking with COSPAN and also directly reduces state spaces. If tight bounds can be provided for variables in a software model, it can often significantly reduce the state space of the resulting formal model that is to be explored by either an explicit state space enumeration algorithm or a symbolic search algorithm. Model checking guarantees the consistency among variable bounds by automatically detecting any possible out-of-bound variable assignments. The annotation language is also used to specify directives for guiding the loop abstraction [13].

Model annotations not only enable transformations, but also are indispensable to translation of continuous or infinite semantic entities in a software model. For instance, in the xUML-to-S/R translation, the information about how to discretize a float type and about the bounds for message buffers of class instances is also provided as annotations.

5.2 Transition Compression

A sequence of transitions in a software model can often be compressed and translated into a single transition in the formal model if verification of the property does not require intermediate states in the sequence. A transition compression algorithm can be generic, i.e., can be applied to many software languages, or can be language-specific, i.e, utilizes language-specific information to facilitate transition compression.

Generic Transition Compression. We use a simple example to illustrate generic transition compression. Suppose a simple program segment is of the form $x = 1;\ x = x+1$. If a property to be checked is not relevant to the interleavings of the two statements with statements from other program segments, to the interim state between the two statements, or to the variable, x, the program segment can be compressed into a single statement $x = 2$ without affecting the model checking result. Similar transition sequences appear in almost all programs in various software specification languages. Detailed discussions on generic transition compression algorithms can be found in [21].

Language-Specific Transition Compression. There will be language-specific opportunities for transition compression in most software specification languages. An illustration of language-specific transition compression in the xUML-to-S/R translation is the identification and translation of self-messages. A self-message is a semantic feature specific to xUML and some other message-passing semantics: a class instance can send itself a message so that it can move from its current state to some next state according to a local decision. (It is assumed that self-messages have higher priority than other messages.) Sending and consuming of a self-message can be translated in a similar way as how sending and consuming of common messages among class instances are translated. This straightforward translation results in several S/R state transitions that simulate sending and consuming of a self-message. We developed a static analysis algorithm that identifies self-messages and translates sending and consuming of a self-message to a single S/R state transition.

5.3 Static Partial Order Reduction (SPOR)

Partial order reduction (POR) [22,23,24] is readily applicable to asynchronous interleaving semantics. POR takes advantages of the fact that in many cases, when components of a system are not tightly coupled, different execution orders of actions or transitions of different components may result in the same global state. Then, under some conditions [22,23,24], in particular, when the interim global states are not relevant to the property being checked, model checkers need only to explore one of the possible execution orders. This may radically reduce model checking complexity.

The asynchronous interleaving semantics of xUML suggests application of POR. POR is applied to an xUML model through SPOR [14], a static analysis procedure that transforms the model prior to its translation into S/R by restricting its transition structure with respect to a property to be checked. For different properties, an xUML model may be translated to different S/R models if SPOR is applied in translation. Application of symbolic model checking to an S/R model translated from an xUML model transformed by SPOR enables integrated application of POR and symbolic model checking.

5.4 Predicate Abstraction

Predicate abstraction [25] maps the states of a concrete system to the states of an abstract system according to their evaluation under a finite set of predicates. Predicate abstraction is currently applied in model checking of software designs in xUML by application of the predicate abstraction algorithms proposed in [17] to the S/R models translated from these designs. It should be possible, however, to implement some forms of predicate abstraction as transformations in translation. Research on application of predicate abstraction to software system designs as they are translated is in progress.

6 Translator Validation and Evolution

Correctly model checking a software model through translation depends on correctness of (1) the conceptual semantics mapping from the source software language to the target formal language, (2) the translator that implements the semantics mapping, and (3) the underlying model checker that checks the resulting formal model. Correctness of a semantics mapping can sometimes be proved rigorously. A proof for the semantics mapping from xUML to S/R can be found in [26]. The translator must be validated to ensure that it correctly implements the translation from the source language to the target language and also the state space reduction algorithms incorporated. The correctness of the model checker is out of the scope of this paper. As the source language and the target language evolve, the translator must also evolve to handle (or utilize, respectively) semantic entities that are newly introduced to the source (or target) language. The translator also must evolve to incorporate new state space reduction algorithms.

6.1 Translator Validation

Testing is the most commonly used method for validating a translator. Testing of a translator is analogous to, but significantly different from, testing of a conventional compiler. Testing of a conventional compiler is most often done by use of a suite of programs which are intended to cover a wide span of programs and paths through the compiler. Testing of a translator from a software specification language to a model-checkable formal language is a multi-dimensional problem. The test suite must be a cross-product of models, properties, and selections of state space reduction transformations. The correctness of a compilation can be validated by running the program for a spectrum of inputs and initial conditions and determining whether the outputs generated conform to known correct executions. While a translated model can be model checked, it is far more difficult to generate a suite of models and properties for which it is known whether or not a property holds on a model. We have a partial test suite for the xUML-to-S/R translation and development of a systematic test suite is in progress. Development of test suites is one of the most challenging problems faced by developers of translation-based model checking systems. We believe this is a problem which requires additional attention.

Recently, there has been progress on formal validation of the correctness of translators. The technique of *translation validation* is proposed in [27], whose goal is to check the result of each translation against the source program and thus to detect and pinpoint translation errors on-the-fly. This technique can improve, however cannot entirely replace the testing approach discussed above since the correctness of translation validation depends on the correctness of the underlying proof checker.

6.2 Translator Evolution

The key to the evolution of a translator is the evolution of the CAR of the translator since the CAR bridges the source software language to the target formal language and connects the translator frontend to the translator backend. Translation from the source language to the CAR is relatively straightforward since the CAR is quite simple. The complexity of the translation from the CAR to the target model-checkable language depends on the complexity of the target language, but the latter are also usually simple and well structured. The transformations conducted on the CAR are much more complex. The principle for the CAR evolution is that the CAR should be kept stable as possible, and existing translation algorithms and state space reduction algorithms should be reused as much as possible. The CAR is extended (1) if there is no efficient way to translate some semantic entities of a new source language, (2) if some semantic entities of a new target language are hard to utilize, or (3) if implementation of new state space reduction algorithms requires introduction of new semantic entities in the CAR.

7 Case Studies Using xUML-to-S/R Translator

The xUML-to-S/R translator has been applied in model checking designs of real-world software systems: a robot control system [28] from the robotics research group at the University of Texas at Austin, a prototype online ticket sale system [29], the TinyOS runtime environment [30] for networked sensors from University of California, Berkeley. The case study [31] on the robot control system demonstrated model checking of nontrivial software design models with the translator. In the case study [32] on online transaction systems, state space reduction capabilities of model transformations in the translator and interactions of these transformations were investigated. The TinyOS case study [33] demonstrated the translation support for compositional reasoning. Co-design and co-verification studies on TinyOS using the translator are in progress.

8 Related Work

Most automatic approaches to model checking of design level software specifications are based on translation. Translators have been implemented for various design level specification languages such as dialects of UML, SDL, and LOTOS [34]. The vUML tool [7] translates a dialect of UML into Promela. The translation is based on ad-hoc execution semantics which did not include action semantics, and does not support specification of properties to be checked on the UML model level. There is also previous work [35,36] on verification of UML Statecharts by translating Statecharts into directly model-checkable languages. The CAESAR system [37] compiles a subset of LOTOS into extended Petri nets, then into state graphs which are then model-checked by using either temporal logics or automata equivalences. The IF validation environment [38] proposes IF [39], an intermediate language, and presents tools for translating dialects of UML and SDL into IF and tools for validation and verification of IF specifications.

The translator architecture presented in this paper extends the architecture for conventional compilers. Similar extensions have been proposed in [37,38]. In these architectures, intermediate representations that have fixed and complete semantics are adopted while in our approach, the CAR does not have fixed and complete semantics. It only specifies semantics of the generally shared semantic entities and for other semantic entities, their semantics are decided when a CAR profile is defined for a specific source language. This enables reuse of translator development efforts while allowing flexible translator development via a customizable intermediate representation.

A recent approach to model checking implementation level software representations is an integrated approach based on abstraction and translation. Given a program in C/C++ or Java, an abstraction of the program is created with respect to the property to be checked. This abstraction is constructed in a conservative way, i.e., if the property holds on the abstraction, the property also holds on the program. The abstraction is then translated into a model-checkable language and model checked. If the property does not hold on the abstraction, the error trace from model checking the abstraction is used to determine if the error is introduced by the abstraction process. If so, the abstraction is refined based on the error trace. The SLAM [3] tool from Microsoft, the FEAVER [6] tool from Bell Labs, and the Bandera [4] tool from Kansas State University are sample projects of this approach. SLAM abstracts a boolean program from a C program, then directly model-checks the boolean program or translates the boolean program into other model-checkable languages. FEAVER abstracts a state machine model from a C program with user help and translates the state machine model into Promela. Bandera abstracts a state machine model from a Java program and translates the state machine model into Promela, SMV, and other model-checkable languages. Many of translation issues identified in our project also appear in the translation phase of these three tools.

9 Conclusions

Translation plays an increasingly important role in software model checking and enables reuse of mature model checking techniques. This paper identifies and formulates issues in translation for model checking of executable software designs. Solutions to these issues are presented in the context of the xUML-to-S/R translator. These solutions can be adapted to address similar issues in translation support for model checking of other design level or implementation level software representations.

References

1. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. Logic of Programs Workshop (1981)
2. Quielle, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. 5th International Symposium on Programming (1982)
3. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. POPL (2002)
4. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Roby: Bandera: a source-level interface for model checking Java programs. ICSE (2000)
5. Havelund, K., Skakkebaek, J.: Applying model checking in Java verification. SPIN (1999)
6. Holzmann, G.J., Smith, M.H.: An automated verification method for distributed systems software based on model extraction. IEEE TSE **28** (2002)
7. Lilius, J., Porres, I.: vUML: a tool for verifying UML models. ASE (1999)
8. Xie, F., Levin, V., Browne, J.C.: ObjectCheck: a model checking tool for executable object-oriented software system designs. FASE (2002)
9. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model Driven Architecture. Addison Wesley (2002)

10. Hardin, R.H., Har'El, Z., Kurshan, R.P.: COSPAN. CAV (1996)
11. Levin, V., Yenigün, H.: SDLCheck: A model checking tool. CAV (2001)
12. ITU: ITU-T Recommendation Z.100 (03/93) - Specification and Description Language (SDL). ITU (1993)
13. Sharygina, N., Browne, J.C.: Model checking software via abstraction of loop transitions. FASE (2003)
14. Kurshan, R.P., Levin, V., Minea, M., Peled, D., Yenigün, H.: Static partial order reduction. TACAS (1998)
15. Holzmann, G.J.: The model checker SPIN. TSE 23(5) (1997)
16. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
17. Namjoshi, K., Kurshan, R.P.: Syntactic program transformations for automatic abstraction. CAV (2000)
18. Kurshan, R.P.: FormalCheck User's Manual. (1998)
19. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. Formal Methods in Software Practice (1998)
20. de Rover, W.P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Non-compositional Proof Methods. Cambridge Univ. Press (2001)
21. Kurshan, R.P., Levin, V., Yenigun, H.: Compressing transitions for model checking. CAV (2002)
22. Godefroid, P., Pirottin, D.: Refining dependencies improves partial-order verification methods. CAV (1993)
23. Peled, D.: Combining partial order reductions with on-the-fly model-checking. FMSD (1996)
24. Valmari, A.: A stubborn attack on state explosion. CAV (1990)
25. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. CAV (1997)
26. Xie, F., Browne, J.C., Kurshan, R.P.: Translation-based compositional reasoning for software systems. FME (2003)
27. Pnueli, A., Siegel, M., Singerman, E.: Translation valdation. TACAS (1998)
28. Kapoor, C., Tesar, D.: A reusable operational software architecture for advanced robotics (OSCAR). The University of Texas at Austin, Report to U.S. Dept. of Energy, Grant No. DE-FG01 94EW37966 and NASA Grant No. NAG 9-809 (1998)
29. Wang, W., Hidvegi, Z., Bailey, A.D., Whinston, A.B.: E-processes design and assurance using model checking. IEEE Computer Vol. 33 (2000)
30. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. ASPLOS-IX (2000)
31. Sharygina, N., Browne, J.C., Kurshan, R.P.: Formal object-oriented analysis for software reliability design for verification. FASE (2001)
32. Xie, F., Browne, J.C.: Integrated state space reduction for model checking executable object-oriented software system designs. FASE (2002)
33. Xie, F., Browne, J.C.: Verified systems by composition from verified components. ESEC/FSE (2003)
34. Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. Computer Networks and ISDN Systems **14** (1988)
35. Gnesi, S., Latella, D., Massink, M.: Model checking UML statechart diagrams using JACK. HASE (1999)
36. Mikk, E., Lakhnech, Y., Siegel, M., Holzmann, G.: Implementing statecharts in Promela/Spin. Industrial Strength Formal Specification Technologies (1993)
37. Garavel, H., Sifakis, J.: Compilation and verification of LOTOS specifications. PSTV (1990)
38. Bozga, M., Graf, S., Mounier, L.: Automated validation of distributed software using the IF environment. CAV (2001)
39. Bozga, M., Fernandez, J.C., Ghirvu, L., Graf, S., Krimm, J., Mounier, L., Sifakis, J.: IF: An intermediate representation for SDL and its applications. SDL-Forum'99 (1999)

Enhancing Remote Method Invocation through Type-Based Static Analysis

Carlo Ghezzi, Vincenzo Martena, and Gian Pietro Picco

Dipartimento di Elettronica e Informazione, Politecnico di Milano
{ghezzi,martena,picco}@elet.polimi.it

Abstract. Distributed applications rely on middleware to enable interaction among remote components. Thus, the overall performance increasingly depends on the interplay between the implementation of application components and the features provided by the middleware. In this paper we analyze Java components interacting through the RMI middleware, and we discuss opportunities for optimizing remote method invocations. Specifically, we discuss how to optimize parameter passing in RMI by adapting fairly standard static program analysis techniques. The paper presents our technique and reports about a proof-of-concept tool enabling the analysis and the subsequent code optimization.

1 Introduction

Two major trends characterize the evolution of software technology during the past decade. On one hand, software applications are increasingly distributed and decentralized. On the other, off-the-shelf components are increasingly used as building blocks in composing distributed applications. The gluing mechanisms that support the assembly of components are provided by the middleware. Although much progress has been achieved in supporting designers while developing distributed applications, it is still true that the level of support provided for traditional centralized software is more mature.

Several techniques are available to optimize code generation for predefined target architectures. Today's challenge, however, is to deal with distributed applications where a conventional programming language is used to develop components and a middleware layer is used to interconnect them. We are not aware of optimization techniques that span over the two domains – i.e., the programming language and the middleware. This is true also when the two are in the same linguistic framework, as in the case of Java and RMI [10].

Providing these optimizations is precisely the goal of our work. We concentrate on parameter passing across network boundaries, when object methods are invoked remotely and parameters are serialized. Serialization may introduce a serious performance overhead for large objects. Furthermore, often only a small part of an object is actually used remotely. In these cases, performance would greatly improve if only the used portion of the object were transmitted.

In this paper we discuss how to achieve this optimization by statically analyzing the bytecode of a Java program, and then using the results to optimize the run-time object serialization. We use fairly standard static analysis techniques. What is new in this paper is the way static analysis is used. Our technique is particularly valuable when off-the-shelf components are used to build a distributed application. In this case, the designer has no visibility of the internals of the components, and therefore many opportunities for hand-optimizing inter-component interactions are necessarily missed. Moreover, since our technique is aimed at reducing the communication overhead, it is particularly useful in bandwidth-constrained scenarios, such as mobile computing.

The paper is structured as follows. Section 2 provides an overview of RMI. Section 3 defines the problem and introduces a reference example. Section 4 describes our program analysis approach. Section 5 reports on a proof-of-concept tool we developed to support our approach. Section 6 discusses limitations and extensions for our technique. Section 7 briefly surveys related work. Section 8 ends the paper with brief concluding remarks.

2 Background

In this section we present the basics of serialization and RMI that are relevant to our work.

2.1 Object Serialization

Serialization is the process of flattening a structured object into a stream of bytes. It provides the basic mechanism to support I/O operations on objects, e.g. to save them on persistent storage, or to transfer them across the network.

Serialization is accomplished in Java by using two I/O streams, `ObjectInputStream` and `ObjectOutputStream`. When an object reference r is written to the latter, the run-time recursively serializes its attributes until the whole graph of objects rooted at r is serialized. In this process, primitive type attributes (e.g. `int`) and `null` attributes are serialized by using a default format. Class descriptors are also inserted in the serialization stream to provide the receiving side with enough information to locate the correct type at deserialization time. Deserialization essentially proceeds backwards, by extracting information from the serialization stream and reconstructing the object graph accordingly. Interestingly, serialization preserves aliases within a single serialization stream.

The aforementioned process requires r and all the other object references to belong to a type implementing the `java.io.Serializable` interface. The programmer, however, retains control over the fraction of the object graph that must be serialized. Although by default all the object attributes are serialized, attributes that are prepended by the `transient` modifier are not. When the object is reconstructed by deserialization, transient attributes are set to the language default for their type.

2.2 Remote Method Invocation

In RMI, a line is drawn between *remote objects* and *non-remote objects*. A “remote object is one whose methods can be invoked from another Java virtual machine, potentially on a different host" ([10], §2.2). Remote objects are defined programmatically by any class that implements the `java.rmi.Remote` interface or a subtype thereof. All the other objects are simply called non-remote objects.

A reference to a remote object can be acquired by querying a lookup service – or *registry* in the RMI jargon. The registry is a process that binds local objects to symbolic names. Remote clients can query the registry by providing a symbolic name, and obtain a network reference to the corresponding object.

A reference to a remote object can also be obtained through parameter passing. Object parameters can be passed in a remote method invocation either by reference or by copy. If the object being passed is a remote object and it has been exported to the RMI run-time, then the object is passed by reference, i.e., it is accessed through the network. Instead, if the parameter is a non-remote object, or a non-exported remote object, it is passed by copy. In this case, however, the type of the object is required to implement the interface `java.io.Serializable`. Primitive types are always passed by copy.

The semantics of parameter passing *by copy* is defined in Java RMI by object serialization. The interplay between serialization and parameter passing, however, slightly complicates the picture. The first issue is aliasing. Since a single serialization stream per remote method invocation is used, references to the same object in the caller are mapped in the callee into references to the same serialized copy of that object. Hence, parameters are not copied independently, as usually happens in parameter passing by copy. The other issue has to do with serialized objects containing references to remote objects. In this case, the behavior of RMI is as follows ([10], §2.6.5):

– If the object being serialized is an instance of `Remote` and the object is exported to the RMI run-time, the stub for the remote object is automatically inserted in the serialization stream in place of the original object.
– If the object is an instance of `Remote` and the object is not exported to the RMI run-time, or the object is not an instance of `Remote`, the object is simply inserted in the serialization stream, provided that it implements `Serializable`.

In essence, this preserves the semantics of object references in presence of distribution. If the object o contains a remote object r in its object graph, the serialized copy of o still accesses the original copy of r on the original node, if r has been exported. Otherwise, r is treated just like any other ordinary object.

3 Motivation and Reference Example

Object serialization can be the source of severe inefficiencies in remote method invocations. In Java – and object-oriented languages in general – objects are often highly structured: composition quickly leads to pretty large object graphs. If the object must be transferred to another host by a remote method invocation, performance may be affected severely. An overhead is introduced in terms of both *computation*, as (de)serialization requires a recursive navigation of the object graph, and *communication*, as large objects result in large serialization streams being transmitted. Most of the existing approaches focus on reducing only the computational overhead [8,1,7,2,12]. They aim at improving the middleware run-time without considering the application code exploiting it and, in particular, how the object is used after deserialization. In this paper, we take the complementary approach of optimizing serialization to reduce communication overhead, based on how serialized objects are used on the server side.

Our goal is to optimize parameter passing for each remote invocation, by serializing only a portion of the object and hence reducing the network traffic. In fact, different invocations may access different portions of the remote objects passed as serialized parameters. The proposed solution consists of (a) performing static analysis to derive information about the portions of serializable parameters that must be transmitted at each call point, and (b) using this information at run-time to optimize serialization.

Let us consider a simple reference example[1], used throughout the paper. A simple print service is provided by the `IPrinter` interface shown in Fig. 1. Clients are expected to invoke the method `print()` by passing the page to be printed as a parameter. Pages are made up of page elements; their interfaces are also shown in Fig. 1. Pages can contain text and/or graphical elements. The methods exported by the `IPage` interface allow one to retrieve either or both. `IPageItem` exports a method `print()`, which is invoked by the receiving `IPrinter` and causes the actual printing of the element on the device.

```
public interface IPrinter
  extends Remote {
  public void print(IPage page)
    throws RemoteException;
}
public interface IPage
  extends Serializable {
  public IPageItem[] getWholePage();
  public IPageItem[] getTextItems();
  public IPageItem[] getGraphicItems();
}
public interface IPageItem
  extends Serializable {
  public void print(PrintStream ps);
}
```

Fig. 1. Interfaces of a simple print service.

In our example, several implementations of `IPrinter` and `IPage` exist, corresponding to different kinds of printing devices and of pages. Sample implementations (`DotMatrixPrinter` and `MixedPage`) are shown in Fig. 2. The client of the printing service, however, ignores the kind of remote printer that is actually being used; it simply invokes the print service. It is up to the server to perform the requested service according to its own capabilities. For example, a dot-matrix printer only prints the textual part of the page (see Fig. 2). Let us consider the case where a composite page is to be printed on a printer that happens to be a dot-matrix printer. Although only textual elements are going to be accessed by the printer, all page elements are serialized and transferred to the server. This is an example where serialization and transmission of a large unused portion of an object generates unnecessary computational and communication overhead, because the server only refers to a portion of the data. The client has no means to avoid this unnecessary serialization – unless information hiding is broken.

We can draw generalized remarks from this example. Often the client has no control over the server's behavior. The server may change over time, due to

[1] The complete source code of the example can be found in [4].

```
public class DotMatrixPrinter
  extends UnicastRemoteObject
  implements IPrinter {
  private PrintStream out = new DotMatrixPS();
  public DotMatrixPrinter()
    throws RemoteException { super(); }
  public void print(IPage page)
    throws RemoteException {
  IPageItem[] text = page.getTextItems();
  if (text != null)
    for (int i = 0; i < text.length; i++)
      text[i].print(out);
  }
}
```

```
public class MixedPage
  implements IPage{
  private IPageItem[]pageItems;
  public IPageItem[]getTextItems(){
    TextItem[] text=
      new TextItem[pageItems.length];
    int j=0;
    for(int i=0;
        i<pageItems.length;
        i++)
    if (pageItems[i]
          instanceof TextItem)
          text[j++]=(TextItem)pageItems[i];
    return text;
  }
  ...
}
```

Fig. 2. Sample implementations of a printer (left) and a page (right).

dynamic binding, and different servers, though presenting the same interface, may differ in their internal behaviors. Internal behaviors are not visible, either because of a deliberate design choice, as in our example, or because they are hard-wired in an off-the-shelf component, whose implementation is responsibility of a third party and hence outside the designer's control.

The next section presents a program analysis technique that enables run-time optimization of remote method invocations. Situations like the one we described can be detected automatically during a static analysis phase, to determine which fields of a given object involved in a remote method invocation are actually used by the target and which are not. The results of analysis can be exploited by automatically generating code that selects the fields to be serialized for each invocation, skipping the serialization of fields unused on the server side. Needless to say, our technique does preserves correctness of the application, i.e., it guarantees that there will never be an attempt to access an object field that has not been serialized.

4 Type-Based Static Analysis

This section describes our static analysis technique in a stepwise manner. We begin by describing the overall analysis strategy, then introduce the notion of concrete graph, which is central to our approach, and conclude by describing the details of the analysis.

4.1 Overall View of the Method

For each remote method invocation `r=o.m`$(p_1, \ldots, p_n)$ and for each serializable parameter $p_i \in \{r, p_1, \ldots, p_n\}$, we identify which attributes of p_i need to be copied through serialization and passed to the remote target object. To achieve this goal, we focus our analysis on the types that can be instantiated through a `new` operation. These types, called *concrete types*, include all primitive types, and do not include any abstract class or interface.

Our analysis technique is structured in the following phases:

1. Given the overall set of types constituting the application, determine:
   (a) the set $\mathcal{R}$ of *remote types*, i.e., types that extend or implement `Remote`;
   (b) the set $\mathcal{S}$ of *serializable types*. This includes primitive types (e.g., `int`) and reference types that extend or implement `Serializable`. The declaration of an array `T[]` causes the insertion in $\mathcal{S}$ of both `T` and the type "array of `T`".
2. Compute the set of *concrete* remote and serializable types, i.e., the subsets $\mathcal{R}_c \subseteq \mathcal{R}$ and $\mathcal{S}_c \subseteq \mathcal{S}$ containing only concrete types.
3. For each class $c \in \mathcal{R}_c$, identify the set $\mathcal{M}$ of methods that can be invoked remotely. This includes all the methods belonging to the interfaces which extend `Remote` and implemented by c.
4. For each remote invocation of a method $m \in \mathcal{M}$, identify the set of parameters $\mathcal{P}_m$ that must be serialized, i.e., those for which at least one dynamic type belongs to $\mathcal{S}_c$.
5. For each parameter $p \in \mathcal{P}_m$, identify the attributes of p for which serialization can be safely skipped.

$$\begin{aligned}
\mathcal{R} &= \{\texttt{IPrinter, DotMatrixPrinter}\}\\
\mathcal{R}_c &= \{\texttt{DotMatrixPrinter}\}\\
\mathcal{S} &= \{\texttt{IPage,MixedPage,}\\
&\quad\ \texttt{IPageItem,IPageItem[],}\\
&\quad\ \texttt{TextItem,TextItem[],}\\
&\quad\ \texttt{GraphicItem,GraphicItem[],}\\
&\quad\ \texttt{int,int[],int[][],char,char[]}\}\\
\mathcal{S}_c &= \{\ \texttt{MixedPage,}\\
&\quad\ \texttt{TextItem,TextItem[],}\\
&\quad\ \texttt{GraphicItem,GraphicItem[],}\\
&\quad\ \texttt{int,int[],int[][],char,char[]}\}\\
\mathcal{M} &= \{\texttt{print}\}\\
\mathcal{P}_{print} &= \{\texttt{aPage}\}
\end{aligned}$$

Fig. 3. The relevant sets for the reference example.

Phases 1-4 are quite straightforward, and can be accomplished by inspecting the code and the inheritance hierarchy. Fig. 3 shows their result for our reference example. Phase 5 is the most complex and constitutes the core of our analysis. In a remote method invocation `r=o.m(`$p_1, \ldots, p_n$`)`, the analysis is complicated by polymorphism. In fact, a parameter p_i of static type T can be replaced at run-time by any subtype of T. The same holds, recursively, for every attribute of T. For each parameter p_i and for each attribute a potentially reachable from it, we must determine whether a is used, and hence it must be serialized, or instead we can safely avoid to do so. The next two sections describe how this can be accomplished.

4.2 Concrete Graphs

Each parameter of a remote method invocation can be associated with one or more descriptors, called concrete graphs. Intuitively, a concrete graph associated with a reference parameter p of type T is a directed multi-graph that represents the type structure of one of the possible instances of p at runtime, according to the class hierarchy. The nodes of the concrete graph are serializable types belonging to $\mathcal{S}_c$. Each edge departs from a node representing the type of an object, ends in the node representing the type of one of the object's attributes, and is labeled with the name[2] of such attribute.

[2] Attribute names must be fully specified, i.e., include the type where they are defined. Hereafter, we use only the attribute label as it is unambiguous in our example.

The concrete graphs for a given parameter are computed through an inspection of the static class hierarchy. Fig. 4 shows the two concrete graphs for a parameter of `MixedPage` type shown in Fig. 2. The two concrete graphs differ in the concrete type of the array attribute `pageItems` of `MixedPage`. The graph in Fig. 4(a) describes the case where an element of the array[3] is of type `TextItem` (a character array). The other graph, in Fig. 4(b), describes the case where the element is instead of type `GraphicItem` (an integer matrix)[4].

Formally, a concrete graph is a tuple $\langle \mathcal{N}, \mathcal{E}, \mathcal{A}, \mathcal{S}_c, type, attr \rangle$. $\mathcal{N}$ and $\mathcal{E}$ are, respectively, the set of nodes and edges of the graph, with $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N} \times \mathcal{A}$, being $\mathcal{A}$ the set of attribute names. $\mathcal{S}_c$ is the set of serializable concrete types. Functions *type* and *attr* represent the object structure:

$$type : \mathcal{N} \rightarrow \mathcal{S}_c \qquad attr : \mathcal{E} \rightarrow \mathcal{A}$$

Function *type* is such that $\nexists n_1, n_2 \mid type(n_1) = type(n_2)$ i.e., each serializable type appears in the concrete graph exactly once. Function *attr* yields the name associated with an edge. For instance, if n_1 and n_2 are the first two nodes of the concrete graph in Fig. 4(a), and e the first edge, then $type(n_1) =$ `MixedPage` and $attr(e) =$ `pageItems`.

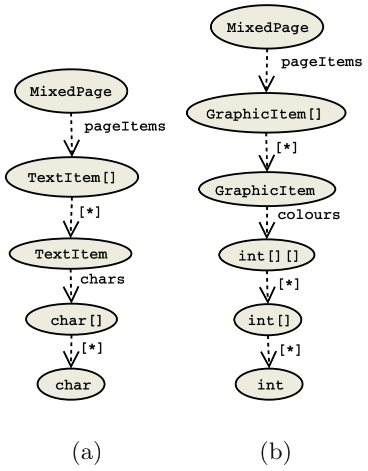


Fig. 4. The concrete graphs of a `MixedPage` parameter.

Intuitively, concrete graphs are used as follows. First, we assume that, for all remote invocations, each serializable parameter has its associated set of concrete graphs. Static analysis is then performed by examining each concrete graph and determining, for each field, if it is used on the receiving side and hence should be serialized[5]. This information is recorded by properly annotating the edges of the concrete graph, and can be exploited at run-time, when the actual dynamic type of each node is known, to determine whether to serialize or skip a given attribute.

4.3 The Analysis in Detail

In this section we provide a detailed description of the core of our technique, i.e., phase 5 of the program analysis described in Section 4.1. To simplify the presentation, we focus on method invocations with a single input parameter and no return parameters. Method signatures with arbitrary arity and types, and encompassing serializable return parameters, can be treated straightforwardly[6].

[3] An edge labelled `[*]` denotes indexing in the array.

[4] Clearly, the array attribute `pageItems` can in general contain any combination of the two.

[5] Clearly, in the case of recursive types only an approximation is possible.

[6] A simple way to deal with multiple parameters is to represent them as attributes of a fake, single parameter. As for the return value, it is sufficient to analyze the client code (instead of the server) using the return value as the parameter driving the analysis.

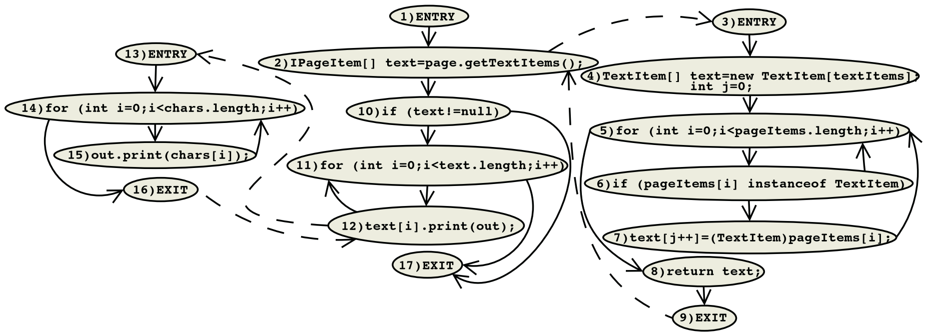

1)ENTRY
2)IPageItem[] text=page.getTextItems();
3)ENTRY
4)TextItem[] text=new TextItem[textItems];
int j=0;
5)for (int i=0;i<pageItems.length;i++)
6)if (pageItems[i] instanceof TextItem)
7)text[j++]=(TextItem)pageItems[i];
8)return text;
9)EXIT
10)if (text!=null)
11)for (int i=0;i<text.length;i++)
12)text[i].print(out);
13)ENTRY
14)for (int i=0;i<chars.length;i++)
15)out.print(chars[i]);
16)EXIT
17)EXIT

Fig. 5. The control flow graph of the methods `DotMatrixPrinter.print`, `MixedPage.getTextItems`, and `TextItem.print`.

Exploiting the Concrete Graph. The analysis of a method `m(p)` starts by building the concrete graphs associated to p. Then, it analyzes the control flow of `m`. As the analysis walks through the body of `m`, it "decorates" each concrete graph of p by keeping track of whether a given attribute can be serialized or not, based on how the control flow has used the attribute thus far. This information is derived incrementally as the control flow is examined, and relies on the definition of two labelling functions mapping each edge of a concrete graph to a boolean value:

$$\mathit{defined} : \mathcal{E} \rightarrow \{\mathit{true}, \mathit{false}\} \qquad \mathit{skip} : \mathcal{E} \rightarrow \{\mathit{true}, \mathit{false}\}$$

The value returned by $\mathit{defined}(e)$ is *true* if the attribute associated with the edge e (i.e., $\mathit{attr}(e)$) has been already assigned a value at a given point in the analysis. The value of $\mathit{skip}(e)$ is *true* if the attribute $\mathit{attr}(e)$ can be safely skipped during the serialization process of the parameter p associated to the concrete graph.

Analyzing the Control Flow. To inspect the control flow of the invoked method, we follow the standard data-flow framework described in [11], which relies on a *control flow graph*, where nodes represent program statements and edges represent the transfer of control from one statement to another. As an example, Fig. 5 shows the control flow graph for the methods `print` in `DotMatrixPrinter`, `getTextItems` in `MixedPage` and `print` in `TextItem` (both invoked by `print` in `DotMatrixPrinter`). The control flow graph of each method starts with an entry node and ends with an exit node. Hence, the overall program control flow can be built out of the method control flow graphs by moving from one control flow graph to the other according to method invocation and termination[7].

Program analysis is carried out by relying on two groups of equations. The first group focuses on a given node in the control flow graph, and defines the relation between the information entering and exiting the node. This group of equations is sufficient to analyze a single path in the given program. However, a node in the control flow graph may have multiple incoming edges that represent

[7] Exception-handling introduces additional implicit control transfers. However, these can be analyzed by using existing techniques (e.g., [13]) in conjunction with ours.

different control flow paths, e.g., due to branches or loops, as shown in Fig. 5. The second group of equations specifies precisely how the information coming from these different sources is merged at the entry point of a given target node n, by defining the relationship between the outgoing information associated to the sources of all the edges insisting on n, and the information effectively entering n. Given these two groups of data-flow equations, the global solution can be computed with a standard *worklist* algorithm (see, e.g., Chapter 6 of [11]).

Object Aliasing. Our analysis is complicated further by *object aliasing*, i.e., the ability of Java to refer to the same object through different references. To determine if an object must be serialized, we must keep track of all the uses of its aliases. The aliasing problem is widely studied and can be tackled by a number of techniques (e.g., [9,14]). Moreover, alias analysis is orthogonal to the type-based analysis described here, and the two can be combined as follows. Given a concrete graph, we first exploit the results of alias analysis to annotate each node of the control flow graph with the alias set associated with each attribute found in the concrete graph. Then, when we "decorate" the edges of the concrete graph, we change the state of an edge not only when an attribute is being modified by a node of the control flow graph, but also when any of its aliases is.

In the sequel, we first describe how the analysis is performed on a single path, by defining how each instruction in the control flow graph of `m` affects the labeling functions *defined* and *skip*. Then, we explain how these functions are "merged" when paths on the control flow graph meet.

Analyzing a Single Path. To simplify the presentation, we assume that the input parameter p in the method invocations `o.m(p)` has a single concrete graph and is serializable, i.e., $p \in \mathcal{S}_c$. Moreover, we assume that all multiple-level reference expression such as `a.b().c()` and `a.b.c` are normalized into a sequence of two-level reference expression of the form `a.b()` and `a.b`, by using additional variables. For instance, `y=a.b().c()` can be split in `x=a.b(); y=x.c()`. In the initial state, $\mathit{skip}(e) = \mathit{true}$ and $\mathit{defined}(e) = \mathit{false}$, $\forall e \in \mathcal{E}$, where $\mathcal{E}$ is set of edges belonging to the concrete graph of p. That is, all the attributes of p are undefined and their serialization can be skipped.

We focus the discussion on a variable y being analyzed in the context of the execution of the given method m, where y is either represented in the concrete graph by some edge e such that $y = \mathit{attr}(e)$ with $e = (n_i, n_j, v)$ and $v = y$, or it is an alias of the variable v represented by e.

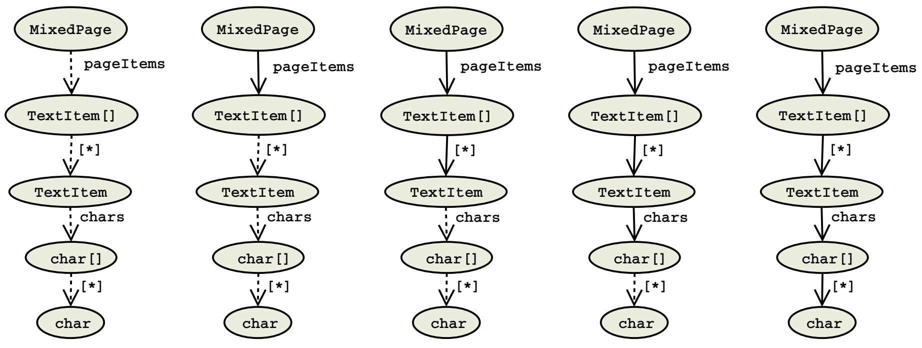
Let us specify how the traversal of a given node of the control flow graph involving y affects the concrete graph, and in particular the labelling of its edges. Variable y can be affected by definitions and uses (in the common meaning of program analysis [15, 6]). *Definitions* of y are statements which assign a new value to y. *Uses* of y are all those situations where y's value (or one of y's attribute values) is used in an expression.

The data-flow equations can then be expressed informally as follows:

– *Definition.* If *defined*(e) is *true* before entering the node of the control flow graph containing the definition of y, nothing needs to be done, since y was already defined and the state of the concrete graph up to date. Otherwise, the value of *defined* is set to *true* for e.
– *Use.* If *defined*(e) is *true* before entering the node of the control flow graph containing the definition of y, nothing needs to be done. Otherwise, the value of *skip*(e) must be set to *false*, since the value of y is needed in the execution of the method under analysis.

Attribute accesses and method invocations are an important kind of use. An *attribute access* to y is in the form[8] `y.x`, where x is an attribute defined in the class of y. It requires to consider, from this point in the analysis on, not only the definitions and uses of y but also those of x, to determine whether it is in turn serializable. *Method invocations* involving y can be either[9] of the form `y.g(...)` or `g(y,...)`. Method invocations can be treated as uses, but they require also method g to be analyzed, by operating on the same concrete graph that was labelled up to the invocation point.

Example. Let us consider the example of Section 3 and the remote invocation of `print` on an object of type `DotMatrixPrinter`, with a parameter of type `MixedPage`. The concrete graphs for this case are shown in Fig. 4, and the control flow graphs are shown in Fig. 5. Let us consider the concrete graph of Fig. 4(a) and walk through all paths of the control flow graph. Fig. 6 shows the result of this analysis as a series of snapshots of the concrete graph as the control flow graph is analyzed. A dashed edge e means that the corresponding attribute can be safely skipped, i.e., *skip*(e) = *true*, while a solid edge means that the attribute must be serialized.



(a) node 1 (b) node 5 (c) node 12 (d) node 14 (e) node 15

Fig. 6. Decorating the concrete graph of Fig. 4(a) while walking through the control flow graphs in Fig. 5. The value of *skip* is *true* for dashed edges and *false* for solid ones.

[8] As mentioned earlier, if y is an array then $y[i]$ is treated as a reference to an attribute.
[9] The `new` instructions can be viewed as a special case of method invocation.

The analysis of the control flow graph begins in node 1 of Fig. 5. Upon entering the first statement of `print` in node 2, the formal parameter `page` (and hence the actual parameter) is used during the invocation of method `getTextItems`. Invocation of this method is analyzed by moving to the entry point of its control flow graph (node 3), but keeping the same concrete graph. The traversal of node 4 leaves the concrete graph unaffected. On the other hand, node 5 contains a use of the attribute `pageItems`, through access to its attribute `length`. The edge corresponding to `pageItems` is then marked as to be serialized, shown with a solid arrow in Fig. 6(b). Node 6 must be considered next. This node is an interesting case since it illustrates how the construct `instanceof` must be handled. Although `pageItems[i]` is an argument of this instruction, this is not a use of the variable. The result of `instanceof` does not depend on the *value* of `pageItems[i]`, but only on its *type*. Moreover, the value of this variable is left unaffected by the execution of `instanceof`. Hence, the traversal of this node leaves the concrete graph unchanged. Note also that in this case we are *forced* to go through node 7 instead of choosing the `else` branch and return to node 5. In fact, choosing the latter path would yield to a violation of the previous assumption about the type of the elements of `pageItems`.

The remaining two nodes of `getTextItems` do not affect the concrete graph directly, but establish the object aliases that enable further changes effected by the other methods. Node 7 establishes an alias between an element of `pageItems` and an element of the local array `text`. Node 8 propagates this alias back to the caller `print`, by returning `text` as a result value. Node 9 brings the control back to `print`, which resumes from node 10. Nodes 10 and 11 do not affect the concrete graph, since they contain only uses of `text`. Node 12 contains a use of an element of `text`, which is potentially aliased to one of `pageItems`. Hence, the corresponding edge in the concrete graph must be marked accordingly, as in Fig. 6(c). Such use is a method invocation, which causes the analysis to move to the control flow graph of `print` (node 13).

The first statement of this method (node 14) contains a use of the array `chars` which, by virtue of aliasing, is an attribute of the element of `pageItem` aliased to the invocation target `text[i]`. Hence, `chars` must be serialized (Fig. 6(d)). Finally, node 15 contains an invocation of the method responsible for printing an element `chars[i]`. Although here we do not show the code of this method, intuitively it relies on the input parameter, which then needs to be serialized, leading to the last and final concrete graph in Fig. 6(e).

According to this analysis, the whole object graph of the parameter must be serialized. In our example this matches intuition, since all the information associated to a text page is actually used by a dot-matrix printer.

Let us examine now what happens if the concrete graph of Fig. 4(b) is considered instead, when walking through the same control flow graphs in Fig. 5. Up to node 6, the analysis proceeds as in the previous case, by requiring the attribute `pageItems` to be serialized. The test in node 6, however, forces us to choose a different path, and return to node 5. In fact, proceeding to node 7 would violate the assumption that the elements of `pageItems` are of type `GraphicItem`. The rest of the analysis proceeds through nodes 5, and 8 to 17. However, since no alias has been established between `text` and some attribute of the concrete graph, the latter remains unchanged: only the edge between nodes `MixedPage` and `GraphicItem[]` in Fig. 4(b) become solid. Hence, the analysis confirms our intuition that serialization of an element of `pageItems` whose type is `GraphicItem` can be safely skipped.

Merging Information from Multiple Paths. What we described thus far is sufficient to analyze methods whose code does not contain branches in the control flow. Otherwise, we need to specify how the information collected through separate control paths is merged when the control paths are rejoined. Such information is the labelling of edges of the concrete graph, i.e., the value returned by the functions *defined* and *skip*. The problem is that an attribute y in the concrete graph may have been recorded as defined ($\mathit{defined}(e) = \mathit{true}, y = \mathit{attr}(e)$) through one control path, and not in another. Even worse, the same attribute may have been deemed necessary to the enclosing method, and hence marked as to be serialized along one path, and marked as to be skipped along another.

Clearly, to preserve a correct program behavior we need to take the most conservative stand. In the aforementioned case we need to preserve, in the node where the control flow rejoins, the values $\mathit{defined}(e) = \mathit{false}$ and $\mathit{skip}(e) = \mathit{false}$. In other words, an attribute is defined in the joining node if it was defined through all of the joining paths, and similarly it can be safely skipped during serialization if it can be skipped through all the joining paths. Formally, if $\mathit{defined}_i$ and skip_i are those computed along an incoming path i, $\forall e \in \mathcal{E}$:

$$\mathit{defined}(e) = \bigwedge_{i=1}^{n} \mathit{defined}_i(e) \qquad\qquad \mathit{skip}(e) = \bigwedge_{i=1}^{n} \mathit{skip}_i(e)$$

Once data-flow equations are given, the analysis is completely defined and the least solution can be computed by the worklist algorithm. The analysis must be performed for each method that can be invoked remotely, for each serializable object parameter, and for each of the possible concrete graphs of such parameter.

5 Prototype

We developed a toolkit to support the approach described in this paper. The overall architecture is showed in Fig. 7. The core component is the *analyzer*, which receives a Java source code as input and outputs information about each remote method invocation, with the corresponding annotated concrete graphs. The output is in binary format for the sake of compactness. The current implementation of the analyzer is built on top of JABA [5], an API supporting program analysis of Java bytecode.

The result of the analysis can be input to one of three tools: The serialization *checker*, which detects all unused fields declared as serialized, as mentioned in

Section 3. The *viewer*, which enables visualization of the analysis output in a human readable format. The *optimizer*, which exploits the analysis results to instrument the source code, by properly redefining serialization. The instrumentation process is non-invasive: it preserves user-defined serialization (when present), and does not affect other uses of serialization (e.g., for storing an object in a file). Details are available in the full technical report [4].

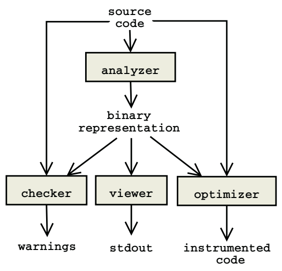


Fig. 7. Tool architecture.

6 Discussion

In this section we discuss improvements and limitations of our technique.

Semi-static Analysis. Program analysis is usually performed statically, and indeed this is the way our approach works, too. The reason is that the computational load is often too high to be placed on the run-time system. Nevertheless, our main goal is to reduce bandwidth utilization. Hence, in some cases (e.g., in mobile environments with low-bandwidth links) it is reasonable to trade computation for bandwidth, and perform some if not all of our analysis at run-time.

The advantage of this approach lies in the accuracy of information about the program that becomes available at run-time. For instance, if the analysis were to be performed right upon a remote method invocation there would be no need to consider *all* the possible combinations of concrete graphs for a given parameter and control flow graphs of the possible servers. Considering the single concrete graph matching the parameter being passed and the specific server target of the invocation would be sufficient. Hence, while on one hand there is a computational overhead to be paid at run-time, this overhead would arguably be significantly smaller than the one to be paid by an entirely static analysis.

Closed vs. Open World. We implicitly assumed that the whole code base of the distributed application is available for analysis, and it is not going to change after the code is deployed. This *"closed world"* scenario holds for a number of distributed applications. RMI, however, was designed to support an *"open world"* scenario where the application code base can change dynamically and seamlessly, by virtue of encapsulation and mobile code. In this case, our analysis would no longer be applicable as is. Let us assume that, in our example, `IPrinter` exports an additional method `getPage` returning the page currently being printed. This method can be invoked by a client C_2, different from the client C_1 that required the page printout. No assumption can be made in general about how C_2 uses the page. For instance C_2 might require the serialization of the entire page as originally stated by the programmer. Now the question is whether the code of C_2 is available at the time of the analysis. If so, the need to serialize all the fields of a page is discovered when the analysis is run on the remote method invocation of `getPage` issued by C_2. Instead, if the code of the clients that may invoke `getPage` is not available, our approach does not work.

We are currently extending our analysis to encompass an open world scenario. This can be achieved by exploiting escape analysis [3], to determine if a given object cannot *escape* the code base known at analysis time. In case it is, the result of our analysis is still valid as is, since an object that has been only partly serialized is never passed outside the boundaries of the analyzed code. Instead, if an object escapes such boundaries no assumption can be made about its use.

7 Related Work

The existing approaches to RMI optimization focus on optimizing the computational overhead of serialization, rather than its bandwidth consumption. Most of these approaches are intended for scientific applications exploiting parallel computing, where computational efficiency is the main concern. To the best of our knowledge no published research has tackled the problem of using program analysis to reduce the traffic overhead of serialization. Thus, no other approach is directly comparable to ours.

Krishanswamy et al. [8] reduce the computational overhead on the client side by exploiting object caching. For each call, a copy of the byte array storing the serialized object is cached to be possibly reused in later calls. Braux [1] exploits static analysis to reduce the computational overhead of an invocation due to the reflective calls needed to discover the dynamic type information. The work of Kono and Masuda [7] relies on the existence of run-time knowledge about the receiver's platform, and redefines the serialization routine accordingly. On the sender side, the object to be serialized is converted directly into the receiver's in-memory representation, so that the receiver can access it immediately without any data copy and conversion. Breg and Polychronopoulos [2] explicitly target homogeneous cluster architectures, and provide a native implementation of a subset of the serialization protocol. Their approach leverages on knowledge about the data layout in the cluster, so that complex data structures are encoded directly in the byte stream by using only a minimal amount of control information. Philippsen et al. [12] integrate various approaches to obtain a slightly more efficient RMI implementation. They simplify the type information encoded in the serialization stream, improve the buffering strategies for dealing with the stream, and introduce a special handling of `float` and `double`. Nevertheless, their optimizations are again closely tied to the parallel computing domain.

8 Conclusions

We presented a novel program analysis technique that aims at optimizing parameter serialization in remote method invocations on a per-invocation basis. The analysis identifies which portion of a parameter is actually used on the receiving side. This information can be exploited to redefine the serialization mechanism and reduce the run-time communication overhead. We implemented a toolkit supporting our approach, and we are currently using it for an empirical evaluation of our method against real-world RMI applications.

Acknowledgments

The work described in this paper was partially supported by the Italian Ministry of Education, University, and Research (MIUR) under the VICOM project, and by the National Research Council (CNR) under the IS-MANET project.

References

1. M. Braux. Speeding up the Java Serialization Framework through Partial Evaluation. In *Proc. of the Workshop on Object technology for Product-line Architectures*, 1999.
2. F. Breg and C.D. Polychronopoulos. Java Virtual Machine Support for Object Serialization. *Concurrency and Computation: Practice and Experience*, 2001.
3. J.-D. Choi et al. Escape Analysis for Java. In *Proc. of OOPSLA'99*, pages 1–19. ACM Press, 1999.
4. C. Ghezzi, V. Martena, and G.P. Picco. Enhancing Remote Method Invocation through Type-Based Static Analysis. Technical report, Politecnico di Milano, 2003. Available at `www.elet.polimi.it/~picco`.
5. Aristotle Research Group. JABA: Java Architecture for Bytecode Analysis. `www.cc.gatech.edu/aristotle/Tools/jaba.html`.
6. K. Kennedy. A Survey of Data Flow Analysis Techniques. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 1, pages 5–54. Prentice-Hall, 1981.
7. K. Kono and T. Masuda. Efficient RMI: Dynamic Specialization of Object Serialization. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 308–315, 2000.
8. V. Krishnaswamy et al. Efficient Implementations of Java Remote Method Invocation. In *Proc. of the 4th USENIX Conf. on Object-Oriented Technologies and Systems (COOTS'98)*, pages 19–36, 1998.
9. W. Landi and B. Ryder. A Safe Approximate Algorithm for Interprocedural Aliasing. *ACM SIGPLAN Notices*, 27(7):235–248, July 1992.
10. Sun Microsystems. Java Remote Method Invocation Specification, 2002.
11. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
12. M. Philippsen et al. More Efficient Serialization and RMI for Java. *Concurrency—Practice and Experience*, 12(7):495–518, 2000.
13. S. Sinha and M.J. Harrold. Analysis of Programs That Contain Exception-Handling Constructs. In *Proc. of Int. Conf. on Software Maintenance*, pages 348–357, Nov. 1998.
14. B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Proc. of the Symp. on Principles of Programming Languages*, pages 32–41, 1996.
15. M. Suedholt and C. Steigner. On Interprocedural Data Flow Analysis for Object-Oriented Languages. In *Proc. of the Int. Conf. on Compiler Construction*, LNCS 641, 1992.

Specification and Analysis of Real-Time Systems Using Real-Time Maude

Peter Csaba Ölveczky[1,2] and José Meseguer[1]

[1] Department of Computer Science, University of Illinois at Urbana-Champaign
meseguer@cs.uiuc.edu
[2] Department of Informatics, University of Oslo
peterol@ifi.uio.no

Abstract. Real-Time Maude is a language and tool supporting the formal specification and analysis of real-time and hybrid systems. The specification formalism is based on rewriting logic, emphasizes generality and ease of specification, and is particularly suitable to specify object-oriented real-time systems. The tool offers a wide range of analysis techniques, including timed rewriting for simulation purposes, search, and time-bounded linear temporal logic model checking. It has been used to model and analyze sophisticated communication protocols and scheduling algorithms. Real-Time Maude is an extension of Maude and a major redesign of an earlier prototype.

Tools based on timed and linear hybrid automata, such as UPPAAL [1], HyTech [2], and Kronos [3], have been successful in modeling and analyzing an impressive collection of real-time systems. While their restrictive specification formalism ensures that interesting properties are decidable, such finite-control automata do not support well the specification of larger systems with different communication models and advanced object-oriented features.

The Real-Time Maude language and tool emphasizes ease and generality of specification, including support for distributed real-time object-based systems. The price to pay for increased expressiveness is that many system properties may no longer be decidable. However, this does not diminish either the need for analyzing such systems, or the possibility of using decision procedures when applicable. Real-Time Maude can be seen as complementing not only automaton-based tools, but also traditional testbeds and simulation tools by providing a wide range of formal analysis techniques and a more abstract specification formalism in which different forms of communication can be easily modeled.

Real-Time Maude specifications are *executable* formal specifications. Our tool offers various simulation, search, and model checking techniques which can uncover subtle mistakes in a specification. Timed *rewriting* can simulate *one* of the many possible concurrent behaviors of the system. Timed *search* and *time-bounded linear temporal logic model checking* can analyze *all* behaviors – relative to a given treatment of dense time as explained below – from a given initial state up to a certain duration. By restricting search and model checking to behaviors up to a certain duration, the set of reachable states is restricted to a finite set, which can be subjected to model checking. For further analysis, the user can write his/her own specific analysis and verification strategies using Real-Time Maude's reflective capabilities.

The time domain may be discrete or dense. Timed automata "discretize" dense time by defining "clock regions" so that all states in the same clock region are bisimilar and satisfy the same properties [4]. The clock region construction is possible due to the restrictions in the timed automaton formalism but in general cannot be employed in the more complex systems expressible in Real-Time Maude. Real-Time Maude deals with dense time by offering a choice of different "time sampling" techniques, so that instead of covering the whole time domain, only *some* moments in time are visited. For example, one strategy offers the choice of visiting at user-specified time intervals; another strategy allows time to advance "as much as possible" before something "interesting" happens. Real-Time Maude's search and model checking capabilities analyze *all* behaviors *up to* the given strategy for advancing time. Search and model checking are "incomplete" for dense time, since there is no guarantee that the sampling strategy covers all interesting behaviors. However, all the large systems we have modeled in Real-Time Maude so far have had a discrete time domain, and in this case search and model checking completely cover all behaviors from the initial state.

Real-Time Maude has been used in some large case studies, including the specification and analysis of a new and sophisticated suite of protocols for reliable and adaptive multicast in active networks, where formal analysis uncovered subtle design errors which could not be found by traditional testing, while independently finding all bugs discovered by testing [5]. Other tool applications include: analyzing a series of new scheduling algorithms for real-time systems and a reliable multicast protocol being developed by the Internet Engineering Task Force, and developing an execution environment for a real-time extension of the Actor model [6].

Real-Time Maude is implemented in Maude [7] as an extension of Full Maude. The current version is a complete redesign of a prototype developed in 2000 [8] and emphasizes new analysis techniques, user-friendliness, and performance. Since most symbolic simulation, search, and model checking commands are implemented by translating them into corresponding Maude commands [9], Real-Time Maude's performance is in essence that of Maude; in particular, the model checking performance is comparable to that of SPIN [10].

The tool – together with a user manual, related papers, and executable example specifications – is available free of charge at `http://maude.cs.uiuc.edu/`.

Specification and Analysis in Real-Time Maude

Real-Time Maude extends the rewriting logic language Maude [7] to specify *real-time rewrite theories* [11]. A Maude module specifies a rewrite theory (Σ, E, R) where (Σ, E) is an *equational theory* specifying the system's state structure and R is a collection of *conditional rewrite rules* specifying the concurrent transitions of the system. A Real-Time Maude specification is a Maude specification together with a specification of a sort `Time` for the time domain, and a set of *tick rules* which model the time elapse in the system and have the form

$\{t\}$ `=>` $\{t'\}$ `in time` u `if` *cond*

where u is a term, possibly containing variables, of sort `Time` denoting the *duration* of the rule, and the terms t and t' are terms of a designated sort `System` denoting the system state. Rewrite rules that are not tick rules are *instantaneous* rules assumed to take zero time. The initial state should always have the form `{`t''`}`, for t'' a term of sort `System`, so that the form of the tick rules ensures that time elapses uniformly in all parts of the system.

The following very simple example models a "clock" which may be running (in which case the system is in state `{clock(`r`)}` for r the time shown by the clock) or which may have stopped (in which case the system is in state `{stopped-clock(`r`)}` for r the clock value when it stopped). When the clock shows `24` it must be reset to `0` immediately:

```
(tmod DENSE-CLOCK is protecting POSRAT-TIME-DOMAIN .
  ops clock stopped-clock : Time -> System [ctor] .
  vars R R' : Time .
  crl [tickWhenRunning] :
      {clock(R)} => {clock(R + R')} in time R'  if  R' <= 24 - R [nonexec] .
  rl [tickWhenStopped] :
     {stopped-clock(R)} => {stopped-clock(R)} in time R' [nonexec] .
  rl [reset] :   clock(24) => clock(0) .
  rl [batteryDies] :   clock(R) => stopped-clock(R) .
endtm)
```

The built-in module `POSRAT-TIME-DOMAIN` defines the time domain to be the nonnegative rational numbers. The two tick rules model the effect of time elapse on a system by increasing the clock value of a running clock according to the time elapsed, and by leaving a stopped clock unchanged. Time may elapse by *any* amount of time less than `24 -` r from a state `{clock(`r`)}`, and by any amount of time from a state `{stopped-clock(`r`)}`. There is no requirement that the specification be "non-Zeno." To execute the specification we should first specify a time sampling regime, for example by giving the command `(set tick def 1 .)` which says that time should advance by increments of `1` in each application of a tick rule. The command `(trew {clock(0)} in time <= 100 .)` then simulates one behavior of the system up to a total duration `100`. The command `(tsearch [1] {clock(0)} =>* {clock(25)} in time <= 100 .)` checks whether the state `{clock(25)}` can be reached from the state `{clock(0)}` in time less than or equal to `100`.

Time-bounded search and model checking are crucial for analyzing systems where the set of reachable states – relative to the chosen strategy for advancing time – is infinite, since the set of states reachable *within the time bound* should be finite. Such a time bound is not needed when the reachable state space, again relative to the chosen "time sampling" strategy, is finite, as in our example. For these cases, Real-Time Maude offers *untime(bounde)d* search and model checking commands, which apply the selected strategy for advancing time but where the search/model checking is not bounded by a time value.

Real-Time Maude's model checker extends Maude's high-performance linear temporal logic explicit-state model checker [10]. Temporal formulas are formed by user-defined atomic propositions and operators such as `/\` (conjunction), `\/` (disjunction), `~` (negation), `[]` ("always"), `<>` ("eventually"), `U` ("until"), etc. Atomic propositions, possibly parameterized, are terms of sort `Prop` and their semantics is defined by stating for which states a property holds. Propositions may be *clocked* in that they also take the elapsed time into account. A module defining the propositions should import the built-in module `TIMED-MODEL-CHECKER` and the module to be analyzed. The following module defines the unclocked propositions `clock-dead` (which holds for all stopped clocks) and `clock-is(`r`)` (which holds if a *running* clock shows r), and the *clocked* proposition `clockEqualsTime` (which holds if the running clock shows the time elapsed in the system):

```
(tmod MODEL-CHECK-DENSE-CLOCK is including TIMED-MODEL-CHECKER .
  protecting DENSE-CLOCK .
  ops clock-dead clockEqualsTime : -> Prop [ctor] .
  op clock-is : Time -> Prop [ctor] .
  vars  R R' : Time .
  eq {stopped-clock(R)}      |=   clock-dead = true .
  eq {clock(R)}              |=   clock-is(R') = (R == R') .
  eq {clock(R)} in time R'   |=   clockEqualsTime = (R == R') .
endtm)
```

The model checking command `(mc {clock(0)} |=u [] ~ clock-is(25).)` checks whether the clock is different from `25` in each computation (relative to the chosen time sampling strategy). The timed model checking command `(mc {clock(0)} |=t clockEqualsTime U (clock-is(24) \/ clock-dead) in time <= 1000 .)` checks whether the clock always shows the correct time, when started from `{clock(0)}`, until it shows `24` or is stopped.

While Real-Time Maude model checking is not complete in general, it *is* a decision procedure for time-bounded temporal properties when the time domain is discrete (which would be case if the imported module `POSRAT-TIME-DOMAIN` were replaced by `NAT-TIME-DOMAIN`) and the instantaneous rules terminate.

References

1. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. Int. Journal on Software Tools for Technology Transfer **1** (1997) 134–152
2. Henzinger, T.A., Ho, P.H., Wong-Toi, H.: HyTech: A model checker for hybrid systems. Software Tools for Technology Transfer **1** (1997) 110–122
3. Yovine, S.: Kronos: A verification tool for real-time systems. Software Tools for Technology Transfer **1** (1997) 123–133
4. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126** (1994) 183–235

5. Ölveczky, P.C., Keaton, M., Meseguer, J., Talcott, C., Zabele, S.: Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. In: FASE 2001. Volume 2029 of Lecture Notes in Computer Science., Springer (2001) 333–347
6. Ding, H., Zheng, C., Agha, G., Sha, L.: Automated verification of the dependability of object-oriented real-time systems. In: 9th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03), IEEE Computer Society Press (2003)
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** (2002) 187–243
8. Ölveczky, P.C., Meseguer, J.: Real-Time Maude: A tool for simulating and analyzing real-time and hybrid systems. In: Third International Workshop on Rewriting Logic and its Applications. Volume 36 of Electronic Notes in Theoretical Computer Science., Elsevier (2000) `http://www.elsevier.nl/locate/entcs/volume36.html`.
9. Ölveczky, P.C., Meseguer, J.: Real-Time Maude 2.0. Submitted for publication. `http://heim.ifi.uio.no/~peterol/RealTimeMaude/rtm-papers.html` (2004)
10. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker. In: Fourth International Workshop on Rewriting Logic and its Applications. Volume 71 of Electronic Notes in Theoretical Computer Science., Elsevier (2002)
11. Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. Theoretical Computer Science **285** (2002) 359–405

A Systematic Methodology for Developing Component Frameworks[*]

Si Won Choi, Soo Ho Chang, and Soo Dong Kim

Department of Computer Science, Soongsil University
1-1 Sangdo-dong, Dongjak-Ku, Seoul, Korea 156-734
{swchoi,shchang}@otlab.ssu.ac.kr, sdkim@ssu.ac.kr

Abstract. Component-based software engineering (CBSE) is being accepted as an effective paradigm for building software systems with reusable components. Product line software engineering (PLSE) is an approach that utilizes CBSE principles to support the economic development of several applications in a domain. Hence, the components should conform to relevant domain standards and they must at least provide common functionality of the domain. Moreover, micro-level variability within commonality should also be modeled in components so that a product member-specific business logic or requirement can be supported through component tailoring or customization. Therefore, the degree of commonality and customizability determines the range of component applicability. In this paper, we propose a systematic approach to identify and model commonality and variability (C&V) and present a methodology to reason about the identified C&V model. With the proposed process and guidelines, components in a product line can better support a larger set of family applications.

1 Introduction

Component-based software engineering (CBSE) has been widely accepted as a new effective paradigm for building software systems with reusable components, consequently reducing efforts and shortening time-to-market. During the last decade, the industry practices of CBSE largely have been producing and utilizing in-house components in order to increase modularity and maintainability beyond object-oriented paradigm. A few case studies of CBSE have been focusing on producing domain common components.

Product line software engineering (PLSE) shares a great deal of CBSE conceptual elements and constructs, but its goal is to economically produce a family of applications by utilizing domain common components. Hence, it is an essential success factor in CBSE to model the common functionalities and features of a domain in order to produce such domain common components. Furthermore, the micro-level variability or alternatives within the commonality should also be modeled in such components so that a product member-specific business logic or requirement can be supported through component tailoring or customization [1]. Therefore, it is fair to state that the degree of commonality and customizability determines the applicability of components in PLSE [2].

[*] This work is supported by the research funding program of Soongsil University.

In this paper, we present a systematic methodology to identify and model the C&V. And, we show how the modeled C&V can be mapped to components and frameworks. In the appendix, we present a case study of building a banking system to show how the proposed methodology can be effectively and practically applied.

2 Related Works

The concept of Object-Oriented Application framework was introduced in order to increase reusability in [3]. Schmidt suggests "plug-in" which one of the different alternatives is plugged into a hot spot in a Framework [4]. Moreover, he proposes a hot spot subsystem as an implemented hot spot. In this study, abstract domain variability is embodied in a hook operation designed with polymorphism and inheritance. However, a larger reuse unit than objects such as component is not considered in this work.

COMO method by Lee [5] suggests a technique to extract common functionality into components by using a clustering algorithm. This method suggests identifying variation points and variants from a family requirement specification. FAST is an early product-line methodology which produces a process pattern for software production [6]. This pattern consists of three main processes; Qualify Domain, Engineer Domain, and Engineer Application. Especially the facility through domain engineering such as application engineering environment and application engineering process can be reused to produce family members rapidly.

Griss [7] suggests using feature model to derive the commonality and variability, where features are clustered into components. This work proposes a four-step process to use features to develop product lines. In addition, the issue of resolving crosscutting features is addressed. However, the process can be better augmented with specific work instructions, artifact templates and traceability framework. Kobra method by Atkinson [2] uses enterprise model, structural model, activity model, interaction model and decision model to model and to specify the variability for framework engineering. A stereotype «variants» is used in these models to indicate the existence of a variation point. A decision model is used to express the variation points for business processes and various diagrams. However, in this work, it is largely unspecified what criteria can be used to determine the existence of variability and how to identify variants and their scopes.

3 The Overall Process

The whole process to model C&V and design component framework consists of five phases and each phase has 2-3 activities as in figure 1. The process has a sequential task flow, but it can be applied iteratively. The details of each phase are specified in sections 3 through 7.

The first phase, *Requirement Normalization,* is to acquire a set of requirements from product members, to identify common vocabulary, and to re-write the requirements using the common vocabulary. This phase is essential to pursue subsequent phases since heterogeneity of different requirements is normalized so that requirements can be compared and C&V can be effectively applied.

The second phase, *Commonality Identification,* is to compare the set of normalized requirements and to identify the common features, i.e. functionality or quality attributes. Once a commonality set is identified, then a *Family Requirement Specification* is constructed from which components are modeled. Variability is a minor difference among family members in their logic or workflow, and it is realized into components so that component consumers can tailor acquired components for their own applications. The third phase, *Variability Identification,* is to identify variability and to design variation points.

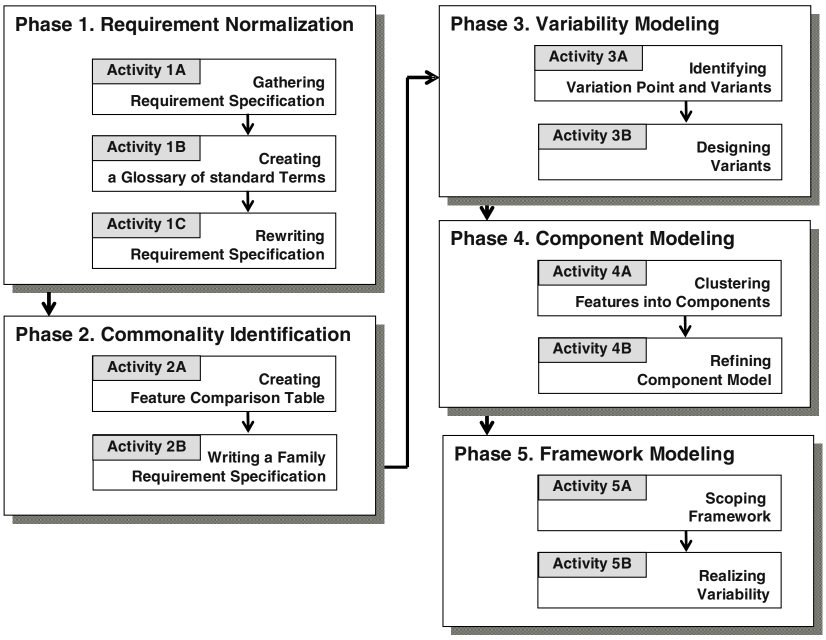

Phase 1. Requirement Normalization
Activity 1A
Gathering Requirement Specification
Activity 1B
Creating a Glossary of standard Terms
Activity 1C
Rewriting Requirement Specification
Phase 2. Commonality Identification
Activity 2A
Creating Feature Comparison Table
Activity 2B
Writing a Family Requirement Specification
Phase 3. Variability Modeling
Activity 3A
Identifying Variation Point and Variants
Activity 3B
Designing Variants
Phase 4. Component Modeling
Activity 4A
Clustering Features into Components
Activity 4B
Refining Component Model
Phase 5. Framework Modeling
Activity 5A
Scoping Framework
Activity 5B
Realizing Variability

Fig. 1. The Overall Process.

The fourth phase, *Component Modeling,* is to cluster features into components and to design preliminary components. Also, variation points are injected into these components and required interfaces are defined. A framework is a large-grained reuse unit which embodies a skeleton architecture, a set of related components and their relationships. An application is created by instantiating this framework. The last phase, *Framework Modeling*, is to identify related components and their relationships, which constitute a framework.

4 Requirement Normalization

This phase consists of three activities; Gathering Requirement, Creating a Glossary of Terms, and Rewriting Requirements.

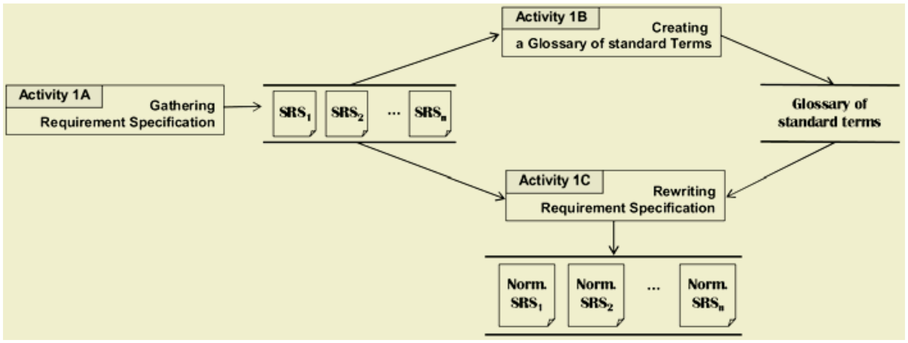

Activity 1A
Gathering Requirement Specification
SRS1
SRS2
...
SRSn
Activity 1B
Creating a Glossary of standard Terms
Glossary of standard terms
Activity 1C
Rewriting Requirement Specification
Norm. SRS1
Norm. SRS2
...
Norm. SRSn

Fig. 2. Requirement Normalization Process and Artifacts.

4.1 Gathering Requirement Specifications

The main goal of product line engineering is to develop reusable assets from which family members' applications can be instantiated in a cost-effective way. Hence, the domain analysis should be applied to a large set of product members, so that the developed components or framework can be widely reused. Activity *1A* is to collect a set of requirements from product members, and these are represented as SRS_i in figure 2. Each requirement may come from a project member, or it can be constructed with a consideration of standard domain logic and knowledge.

4.2 Creating a Glossary of Standard Terms

One of the first obstacles in PLE is the heterogeneity of the requirement specifications gathered from several product members. The heterogeneity is largely appeared as inconsistency and ambiguity on terminology and concepts used in the requirements. A single term may have different meanings among product members, and several different terms among product members may have a single meaning.

Since the components used in PLSE should provide the standard or common features among product members, component producers must compare the set of requirements and identify a commonality set. But, this heterogeneity makes the comparison of various requirements impractical and inefficient. Hence, the set of requirements must be normalized using standard terms and concepts.

Activity *1B* is to derive a glossary of standard terms from the set of requirements gathered during activity 1A. In order to facilitate the process of identifying standard terms, we use a term comparison table as in table 1. We first grouping similar terms, T(1,1), T(2,1), … T(n,2) from the requirements and identify the most commonly used or standard term.

Table 1. Term Comparison Table.

Member / Category	M_1	M_2	…	M_n	Common Term
	T(1,1)	T(2,1)	…	T(n,2)	
	T(1,2)	T(2,3)	…	T(n,4)	

Once a term comparison table is constructed, we create a glossary of standard terms which is referred by all further activities and it provides a common definition of terms used.

4.3 Rewriting Requirement Specifications

By using the glossary of standard terms, we re-write the requirement specifications of product members. This will make it easier to compare the features among product members and to identify the commonality and variability since the revised requirement specifications will be expressed in all standard terms. However, if the requirement specifications from the product members are relatively homogeneous and there exists only minor difference in the terms used, then this activity can be omitted. In figure 2, a *Norm SRSi* is a re-written requirement specification, called normalized requirement specification.

5 Commonality Identification

During the first phase, requirement normalization, we normalized a set of heterogeneous requirements and domain knowledge. The second phase, commonality identification, will compare several requirements of product members to derive a set of common features among them. This common set will be used as the basis to determine the scope of candidate components and frameworks.

5.1 Creating a Feature Comparison Table

In order to effectively compare the set of requirements, we use a *Feature Comparison Table* as in table 2. For the ‘n’ members of the PL, their features are compared for potential commonness in this table.

Table 2. Feature Comparison Table.

Features	Product Members				Degree of Commonality	Rules Applied	Decision (Y/N)
	M_1	M_2	…	M_n			
F_1	√	√		√			
F_2		√		√			
…	√	√	√				
F_m	√		√	√			

A *feature*, as in PLE [7], is characterized by functionality and quality attributes. In many cases a feature maps to functionality. A PL has a set of features; F_1, F_2, .., F_n where F_i is a specific feature in PL. The first column lists all the features, F_1, F_2, ..., F_m, found in a product line. Earlier this set was defined as a union of the requirements.

A difficulty in creating this set in practice is to apply a consistent degree of granularity to all the features. This is because the granularity of features in a member's requirement may be different from that of other member's requirement. Hence, one should come up with an appropriate granularity level when there is a dispute on different granularity levels on a single feature. This granularity normalization can often be done with the intensive participation of domain experts.

Some of the features will be common among members while others are non-common, i.e. specific only to one or a few members. In the columns for product members, we express how each feature, F_i, is applied to each member, M_j. If F_i is applied to M_j, i.e. the member M_j requires F_i, a check mark is given. In deciding the applicability of features, we only consider the overall functionality and quality attributes at macro level.

In the column of 'Degree of Commonality', we specify a metric for each feature as (Number of check marks) / (Total Number of Members). This metric gives only an approximate degree of commonness for the given feature since one member's requirement may be more valuable or dominant than other members due to the different representation of the member in the domain.

In the column of 'Rules Applied', we specify the rules that have been applied in making decisions on whether or not each feature is included in the commonality set. In making decisions, we consider several factors; the degree of commonality, business influence of each member, sponsorship such as funding, and the degree of standardization. Although defining a set of precise rules for this decision making is not feasible, we propose the following candidate rules as a starting point;

i) If the degree of commonality is 100%, it is included in the set.
ii) If the degree of commonality is near 100% and there are some influential member s who require the feature, then it is included in the set.
iii) If the degree of commonality for a feature is relatively lower than those of other fe atures and there is no influential member who requires the feature, then it is not in cluded in the set. If a member is a key player in the domain in terms of business sc ale and market share or a client who pays the cost of developing reusable assets, th en it is included in the set.
iv) Other case which lies between the cases ii) and iii) should be judged with the dom ain knowledge, members' influence and business issues such as marketability.
v) If the feature is an essential intrinsic or standard functionality in a domain, then it can be included in the set with careful judgment.

The last column of 'decision' is about whether the feature should be included in the commonality set or not. The decision on whether or not a feature is common is mostly made based the above rules, but other business factors or domain constraints can also be considered.

5.2 Writing a Family Requirement Specification

Based on the comparison, we can now summarize the common features in *Commonality Specification Table* as in table 3. The first column is the identification number of each feature, and any reasonable number scheme can be used. The second column is for the names of features, and the third column is the description of features.

Table 3. Commonality Specification Table.

Feature ID	Feature Name	Description
CF_1		
CF_2		
. . .		

By using the common features in this table, we create a *Family Requirement Specification* which will be used in later phases as the reference for building components and frameworks. Also, the information in this table can be provided to component consumers so that they understand what services the components provide and what to expect from the reusable assets.

6 Variability Modeling

Through *Commonality Identification* activity, we have identified a common set of features that should be realized in components. However, a careful examination on a common feature often reveals a minor variation on logic or workflow. This is called variability within a commonality. By realizing this variability in developing components, the range of applicability and so reusability of components can be greatly increased [8].

6.1 Identifying Variation Point and Variants

A variation point of a feature in PLE is an identifier of a hot spot where the variability among different product members occurs [2]. In order to effectively model the variability, we use the *Variability Identification Table* as in table 4.

Table 4. Variability Identification Table.

Common Features	Variation type	Product Members					Range
		M_1	M_2	M_3	…	M_n	
CF_1	Logic	$V_{1.1}$	$V_{1.2}$	$V_{1.1}$		$V_{1.1}$	2
CF_2							
CF_3	Work-flow	$V_{3.1}$	Open	$V_{3.2}$		$V_{3.2}$	2+Open
. . .							
CF_m	Logic	Open	Open			Open	Open

In the second column, the type of a variation point is specified. In both CBD and PLE, a variation point can be in a form of logic and workflow in practice. The *logic* in this context consists of several steps which correspond to program statements once implemented. Hence, a variation point of *logic* describes a set of different algorithms for a system or business operation.

A variation point of *workflow* describes a set of different message flows for a system or business operation since a workflow is typically realized by a sequence of message invocations possibly over multiple objects or components. An example can be in a banking system product line that there can be different workflows, i.e. procedures, to evaluate and approve a loan application.

In the next set of columns, M_1, M_2, … , M_n, all possible instances of a logic or workflow variation point for CF_i, i.e. variants, are identified and specified. A variant, $V_{i,j}$ is jth variant of ith common feature, CF_i. In the first row for CF_1, the variant of M_1 is specified as $V_{1,1}$. If the variant for M_2 is not same as $V_{1,1}$, then M_2 is given a new variant ID. In the row, the M_3 is shown to have the same variant as M_1's. By repeating this procedure, all the possible variants for each variation points are identified.

A key problem in completing this table is to decide whether variability exists between any pair of product members. That is, what exactly is the difference between $V_{i,p}$ and $V_{i,q}$? We propose the following decision rules based on the elements of post condition, input domain, output range, and realization algorithms;

i) *If the post conditions of* CF_i *for two members are different, then there exists a variation.* A post condition is specified on the result of CF_i, and if the post conditions for two members are different, it implies that the logics or workflows for two members are different in some degree.

ii) *If input domains or output ranges of CF_i for two members are different, then there exists a variation.* An input domain for a feature is a set of all possible input values and/or types, and an output range is the set of all possible values and/or types generated by the feature. If input domains are different, then the logics or workflows to manipulate the input values will be different. Similarly, if output domains are different, then there must be different algorithms or workflows to produce different sets of output values.

iii) *If the realization algorithms for CF_i can be determined at this stage, and the algorithms for two members are different, then there exists a variation.* In some cases, the realization algorithms are not available until a later phase. But in some other cases, such algorithms can be available as a pre-fixed requirement. In this case, two algorithms can be compared to determine the existence of variability.

iv) *If none of the above rules can be applicable to CF_i, then other fact*ors such as pre-condition, invariants, and semantic description should be considered for a comparison.

The last column of *Range of Variation* specifies the total number of variants identified by decision rules. If this number is equal to 1, then there is no variability for the common feature. If this number is greater than 1, then CF_i has a variation point and a set of variants. If a variant for a variation point is unknown, i.e. open, then, it is marked with 'open'. If some variants are known and also some variants are open, then we put the total number of variants followed by the 'open'.

6.2 Designing Variants

From *Variability Identification Table*, we decide and specify how to realize the variants for each variation point. We use *Variability Range Table* as in table 5.

Table 5. Variability Range Table.

Variable Features	Type of Variation	Set of Variants	Open/ Closed	Default	Description
CF_1	Logic	$\{V_{1.1}, V_{1.2}\}$	Closed	$V_{1.1}$	
CF_3	Workflow	$\{V_{3.1}, V_{3.2}\}$	Open	$V_{3.2}$	
. . .					
CF_m	Logic	{ }	Open	None	

The first column lists only the common features that contain variation points, and so this information can be copied from the *Variability Identification Table*. The second column specifies the type of variation type which is also available in the *Variability Identification Table*. The third column, *List of Variants* lists all the variants identified. The next column, *Open or Closed*, specifies a binary value to indicate whether the list of variants identified is complete, i.e. closed, or expandable in the future, i.e. open. Depending on this openness, different implementation techniques can be adopted. The next column, *Default*, specifies a default variant among the list of variants, so that the components can be consumed without tailoring process if the default variant is needed during application engineering.

7 Component Modeling

7.1 Clustering Features into Components

During the previous activities, a set of common features and its variability scope have been identified. Based on this C&V model, conceptual components are designed. There is no mechanical procedure to identify components, but we apply the following guidelines to help clustering *related* features into components as in figure 3. Two features CF_i and CF_j are related if following conditions hold;

i) Features CF_i and CF_j are related if they belong to a same functional category as de
fined by clients. This functional category may be based on system, module, functi
onal classification and deployment classification. Typically, clients have in-depth
domain knowledge and possess a functional classification scheme of features acco
rding to their domain knowledge. And, so if two features belong to a same functio
nal category defined by clients, then these are said to be related.

ii) Features CF_i and CF_j are related if they use common data or information. A featur
e is a small-grained functionality required by clients, and each feature uses a set of
data elements or information. If two features use exactly or mostly same set of dat
a or information, they are grouped into a component.

iii) Features CF_i and CF_j are related if there is some strong degree of dependency bet
ween the features. A dependency in OOP and CBD is a method invocation relation
ship. Two features with dependencies should be clustered into a single component
in order to minimize the coupling.

iv) Features CF_i and CF_j are related if they belong to the system layer and they proces
s related system operations or transactions. Typically a feature that processes syste
m operations belongs to a system layer, and the set of such features should be grou
ped into a single component. Hence, these features can be distinguished from the f
eatures that manipulate persistent data or objects.

v) Features CF_i and CF_j are related if they belong to the business layer and they mani
pulate persistent data or objects. In contrast to iv), these features belong to busines
s layer, and so they can be distinguished from the features that process system ope
rations or transactions.

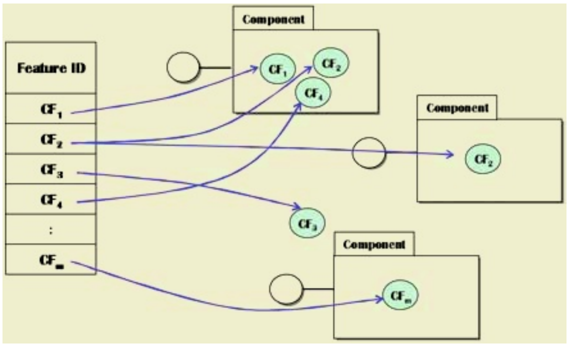


Fig. 3. Grouping Features into Components.

A feature is a system behavior exposed to clients, and so it tends map to a system component. However, a feature can map to a business component if the nature of the feature is mostly CRUD data manipulation. Hence, it is common in practice that a feature maps to a system component, which in turn invokes operations of a business component yielding an inter-component dependency.

7.2 Refining Component Model

The above set of decision rules is neither definitive nor complete since this grouping process largely depends on the domain knowledge and there can be exceptions to the rules. Two problematic cases that could be generated by applying the decision rules are unassigned features such as CF_3 and features appearing in multiple components such as CF_2 as shown in figure 3.

An unassigned feature can be grouped into a component that is the closest to the feature. If there are a large number of unassigned features, then they are grouped into a *utility* component. If a feature appears in multiple components, then we use the following decision rules;

i) If the functional nature of the feature is mostly information retrieval rather than inf
ormation update, then the feature is duplicated into multiple components for conve
nience and efficiency.

ii) If the functional nature of the feature is mostly information update rather than retri eval, then assign the feature to one component which uses the feature most intensi vely. And, let other components access this component with the feature through int erface. In this way, we reduce data/state inconsistency problem with duplicated fea tures while providing ways to access the feature.
iii) If the feature has the characteristics of the case ii), but it is not feasible to find a co mponent which will contain the feature, then, group such features into a common component. And, let other components access this feature through the interface of this common component.

Figure 4 shows that the unassigned feature and figure appearing in two components are re-configured according to the decision rules.

8 Framework Modeling

8.1 Scoping Framework

Once the components are identified, then frameworks are designed by grouping related components. A framework is semi-completed application and hence its granularity is larger than components. Therefore, in most cases, we have a single framework for a product line but there can be multiple frameworks in some cases. We use the following guidelines in determine whether two components are *related*;
i) Two components are related if both components are required to constitute a sub-sy stem. This is because a framework embodies a skeleton architecture of sub-system or a whole system.
ii) Two components are related if both components map to structural elements of a st able and skeleton application architecture of the product line.
iii) Two components are related if there is a dependency or association relationship. In ter-component relationships should be captured within a framework since a frame work is highly cohesive large-grained reuse unit.

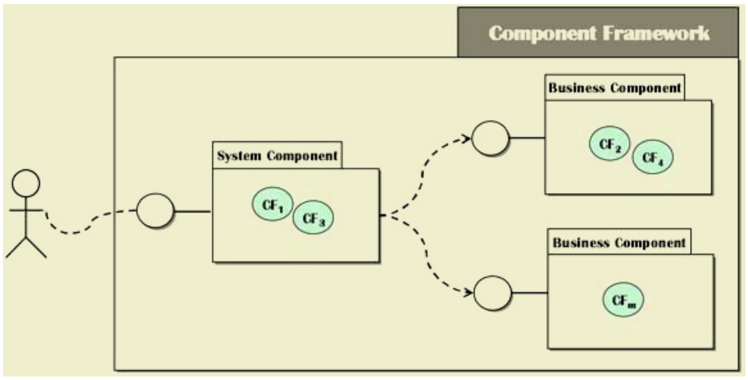


Fig. 4. Grouping Components into Framework.

Figure 4 shows that the three components are grouped into a single framework. The system component interacts with clients whereas business components act upon the invocation by system components through mediate pattern.

8.2 Realizing Variability

Once components are clustered into a framework, then we project the variability information specified in table 5 into frameworks. Typically variations points are realized inside components, and methods to set variants for variation points are defined in a *required interface* as shown in figure 5.

Tailoring components is different from invoking component methods in several ways. Tailoring components is typically done once per deployment or installation whereas invoking component methods are frequently made at run-time. The variant set during tailoring process will remain persistently within the component whereas actual parameters passed through method invocation are transient, i.e. short lived. Hence, a framework or components must maintain the variant set during tailoring process as persistent information. This is shown as *'CurrentVariant'* persistent attribute in figure 5.

If a variation point has a *Closed* scope, then it is tailored by using *Select( )* method. This method will take variants required by each application and store them persistently. If a variation point has a completely *Open* scope, then it is tailored by using *PlugIn( )* method. This method will take a reference to an external function, object or component, and invoke the method provided by the plugged in entity. If a variation point has a partially *Open* scope and some variants are known, then we use both *Select( )* method and *PlugIn( )* method.

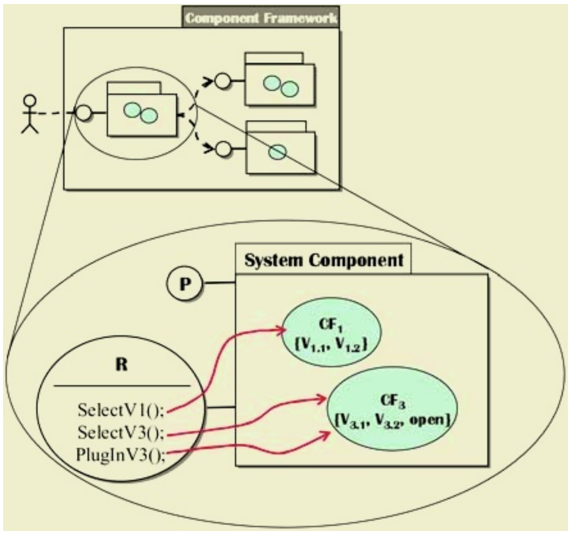


Fig. 5. Variation points projected into Framework.

As shown in figure 5, the CF_1 has a logic variation point of *Closed* scope and its tailoring method in the required interface is defined as *SelectV1();*. In the case of CF_3, the workflow variation point has an *Open* Scope with two known variants; $V_{3.1}$ and $V_{3.2}$. Therefore, two tailoring methods are used; *SelectV3()* and *PlugInV3()*. If one of the two known variants is required for a product member, then the *SelectV3( )* method is invoked. If the built-in variants, i.e. workflows, cannot be applied to the product member, then a plug-in object will be passed through *PlugIn( )* method.

9 Traceability

The process proposed in this paper includes 11 activities, and each activity produces one or two artifacts as summarized in figure 6. We now show the traceable items between pairs of artifacts in figure 8. The arrow between a pair of artifacts indicates the transformation direction, and the expression *Item1*→ *Item2* on arrows indicates the *Item1* is a source artifact from which a target *item2* is derived. And, so the target artifact can be traced to the source artifact through the transformation items. For *variants* in the artifact *Variability Range Table* are source items from which *variation design* and *required interface* in a *framework* are derived as in the figure.

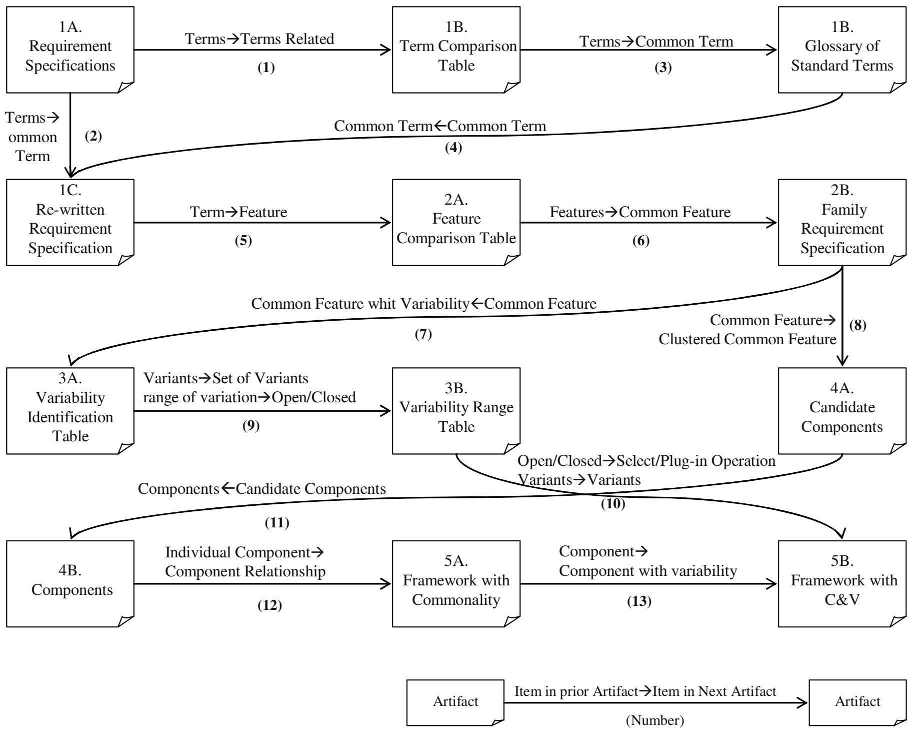

1A. Requirement Specifications
Terms→Terms Related
(1)
1B. Term Comparison Table
Terms→Common Term
(3)
1B. Glossary of Standard Terms
Terms→ ommon Term
(2)
Common Term←Common Term
(4)
1C. Re-written Requirement Specification
Term→Feature
(5)
2A. Feature Comparison Table
Features→Common Feature
(6)
2B. Family Requirement Specification
Common Feature whit Variability←Common Feature
(7)
Common Feature→ Clustered Common Feature
(8)
3A. Variability Identification Table
Variants→Set of Variants
range of variation→Open/Closed
(9)
3B. Variability Range Table
4A. Candidate Components
Open/Closed→Select/Plug-in Operation
Variants→Variants
(10)
Components←Candidate Components
(11)
4B. Components
Individual Component→ Component Relationship
(12)
5A. Framework with Commonality
Component→ Component with variability
(13)
5B. Framework with C&V
Artifact
Item in prior Artifact→Item in Next Artifact
(Number)
Artifact

Fig. 6. Traceable Items among Artifacts.

Each mapping in figure 6 specifies a transformation of a source item onto a target item, and a set of rules, called *traceability rules,* can be defined to show the validity of transformation. Due to the paper length, we only show rules for the mapping (10) Variability Range Table to Framework with C&V as an example.

– Rule 1. Each variant with Closed feature is mapped to a customization method Select<VP name> (VariantType v) in Required interface. The VariantType must be a datatype that specifies a set of all possible variants, so that an argument of VariantType can be passed by component clients.
– Rule 2. Each variant with Open feature is mapped to a customization method PlugIn<VP name> (PlugInObject p) in Required interface. The PlugInObject must be a class type that models a set of all variant objects that can be passed.
– Rule 3. A feature with both known variants and Open range must be mapped to two customization methods; a Select<VP name>( ) for known variants and a PlugIn<VP name>( ) for Open variants.

Similarly, set of traceability rules can be defined for other mappings.

10 Conclusion

Product line software engineering is a practical framework that utilizes CBSE principles in order to support the economic development of a set of applications in a domain. Hence, the components used in PLSE should conform to relevant domain standards or they must at least provide common functionality of a domain. Also, the variability should be modeled in components so that a product member-specific business logic or requirement can be supported through component tailoring or customization.

In this paper, we proposed a 5-phase process to identify and model the commonality and variability (C&V) and present a framework to reason about the identified C&V model in order to enable effective implementations of PLSE components. Activities within a phase are given a set of instructions and artifact templates. The whole process has been applied to a case study of banking domain. In addition, the traceability among artifacts and guidelines to enforce the traceability were given. With the proposed process and guidelines, the C&V can be systematically modeled into component framework, and the quality of delivered frameworks can be increased by applying traces using the proposed traceability foundation.

References

1. D'souza, D., *Objects, Components, and Frameworks with UML*, Addison Wesley, 1999.
2. Atkinson, C., et al., "Product Line Concepts", Chapter 14 of *Component-based Product Line Engineering with UML*, Addison Wesley, 2001.
3. Fayad, M. and Schmidt, D., "Introduction," *Communications of the ACM*, Oct. 1997.
4. Schmidt, H., "Systematic Framework Design by generalization," *Communications of the ACM*, Oct. 1997.
5. Lee, S., Yang, Y., Cho, E., and Kim, S., "COMO: A UML-Based Component Development Methodology", *Proceedings of Asia-Pacific Software Engineering Conference (APSEC99)*, Takamachu, Japan, pp. 54-61, Dec. 7-10, 1999.

6. Weiss, D. and Chi, T., *Software Product-Line Engineering: a Family-Base Software Development Process*, Addison Wesley, 1999.
7. Griss, M., “Product-Line Architectures,” Chapter 22 of *Component-Based Software Engineering*, Addison Wesley, 2001.
8. Kim, S., and Park, J., “C-QM: A Practical Quality Model for Evaluating COTS Components,” *Proceedings of International Association of Science and Technology for Development (IASTED) International Conference on Software Engineering*, Innsbruck, Austria, pp.991-996, Feb. 10-13, 2003.
9. Kim, S., “Lessons Learned from a Nationwide CBD Promotion Project,” *Communications of the ACM*, Oct. 2002.

Automating Decisions in Component Composition Based on Propagation of Requirements

Ioana Şora[1], Vladimir Creţu[1], Pierre Verbaeten[2], and Yolande Berbers[2]

[1] University Politehnica of Timisoara, Department of Computer Science and Engineering, Bd. V. Parvan 2, 1900 Timisoara, Romania
ioana@cs.utt.ro

[2] Katholieke Universiteit Leuven, Department of Computer Science
Celestijnenlaan 200A, 3001 Leuven, Belgium

Abstract. Automatic component composition is a way to achieve self-customizable systems that are able to adapt themselves through structural configuration to changing conditions in their environment. In this paper, we propose an automatic composition strategy for multi-flow architectures with hierarchically composable components. Our composition strategy takes automatic decisions for the composition of a target that is specified through a set of required properties imposed over its given structural constraints. The composition decisions are taken knowing the properties provided by individual available components. Properties characterize functional or non-functional aspects of a component. The composition strategy is driven by a mechanism of propagation of required properties, detailed in this paper.

1 Introduction

Component-based development is a proven approach to manage the complexity of software and its need for customization. An important challenge is to build new systems that provide certain properties, by systematically composing reusable components. Our research approaches the problem of component composition from the point of view of the decisional question: how to decide what components will be deployed and what collaborations will be between them?

The need for rigorous strategies for compositional decisions appears particularly in circumstances when the composition decision must be a machine decision, as it is the case when automatic component composition is used as a means to realize self-customizable systems. Our work addresses self-customizable systems that are able to adapt themselves to their evolving runtime environment. Such automatic software composition is based on a compositional model that comprises:

– A *component description scheme and formalism*. This establishes what information is needed to be known about the components in order to make composition decisions.

– A well defined *requirements driven composition strategy*. This establishes the rules for selecting the necessary set of components and determining their integration, based on the available information about the components.

Many approaches tackle composition in domain-specific ways. We argue that compositional models should be architectural style specific and independent from application domains, to create a premise for generic solutions. We have developed a compositional model for multi-flow architectures based on composable components. It comprises a component description scheme for hierarchically composable components with its description language CCDL (introduced in [ŞVB03]) and a requirements driven composition strategy, which is described in this paper.

The composition strategy implements rules for finding a component composition with desired properties, based on the properties of individual components described according to the CCDL scheme ([ŞVB03]). Supporting unanticipated compositions (in terms of deployed components and structure) is a main objective of our composition approach. This paper presents the principles of our composition strategy, introducing the mechanism of propagation of requirements as its driving element.

The remainder of this paper is organized as follows: Section 2 presents briefly our architectural component model as the premise of our work on automatic composition. Section 3 details the mechanism of propagation of requirements and Section 4 presents the composition strategy. Finally, Section 5 discusses our research in the context of related work, while the last section summarizes the conclusions.

2 Architectural Model and Composable Components

This section presents briefly the main concepts of our component model.

A software system is viewed as a set of components that are connected by connectors. A software component is an implementation of some functionality, available under the condition of a certain contract, independently deployable and subject to composition [Szy97]. Moreover, a component in our approach is also an architectural abstraction.

We consider that the system architecture reflects *interaction* relationships among the components. A component has a set of *ports* for the interaction with the rest of the system. A port is "a logical point of interaction between the component and its environment" [AG97]. Distinction is made between input and output ports. In our approach all components are considered plug-compatible in the sense that an input port can be connected to an output port.

Our current work on composition investigates systems that have a *multi-flow architecture*. The concept of *flow* corresponds to the data-flow relation among pairs of ports. A flow has parts where it is internal to a component (from an input to an output port of that component) and parts where it is between two components (a connection). We define the multi-flow architecture as a variation of the pipes-and-filters architecture, where the architecture of a system is completely defined by the dataflow relations (the "flows" in our terminology).

The flows are fixed, while the positions of components on these flows are not important. Components must just fit on the fixed flows. For every component, the internal flows must be known so that they can be integrated in such a flow architecture.

Components can be simple and composed. A simple component is the basic unit of composition, has one input port and one output port. A composed component is an aggregation of several other components, it may have several input and output ports. Multi-flow architectural style applies to the internal configuration of composed components.

A component's contract specifies the services provided by the component and the obligations of its clients and environment. In our approach, contracts are expressed through sets of required-provided *properties*. A component property is a fact that is known about the component. In our approach, a property is expressed through a name from a vocabulary set and may have attributes or refining subproperties. The name of the property is treated in a semantic-unaware way: this means that a match between required and provided properties is established by matching names and attributes. Provided properties are associated with components as a whole, requirements are associated with ports.

An important element of our approach is that composed components are also "first class" components, they have their own properties and contractual interfaces. A composed component as a whole is always defined by its own set of provided properties, which expresses the higher-abstraction-level features gained through the composition of the subcomponents. The vocabulary used to describe the own provided properties of a composed component is distinct from the vocabulary deployed for describing the provides of its subcomponents. This abstraction definition must be done by the designer of the composed component.

The internal structure of a composed component is mostly not fixed, these components are *composable* in the limits of certain structural constraints. These structural constraints ensure the preservation of the identity of the composable component. As they have been introduced in [ŞVB03], the structural constraints are flexible guidelines for future compositions of the internal structure but not a full configuration description. Structural constraints of a composable component are expressed through: the set of internal flows, the properties that must exist on these flows, and possibly the order relationships between these properties. The structural constraints are a solution that balances between the need to support unanticipated customizations of the internal structure of a composable component and the need to preserve the properties that determine the identity of the composable component.

An important strength of our approach is that it does not limit the customization of composable components to filling in a given structure with right implementations. It is possible that new components, which can provide further enhancements or customizations for the composed component, are discovered. The insertion of these new subcomponents is permitted anywhere on the existing flows, as long as their component descriptions do not contradict existing requirements (structural constraints of the composed component or requirements of the components already present on that flow). The composition strategy automatically decides which subcomponents to deploy on the internal flows of the composed target, starting from the requirements imposed by the client and complying with the structural constraints of the target. This strategy is based on a mechanism of propagation of requirements that will be detailed in the next sections.

3 The Mechanism of Propagation of Requirements

The operation of matching required properties of one component with provided properties of other components can be a complex process. The complexity resides in the fact that interactions of properties cannot be isolated to pairs of interacting components, but most often there are large groups of components with transitive interaction relationships between them. In order to manage the complexity of such situations we define and use a mechanism of propagation of requirements. This is discussed here, starting in subsection 3.1 with a simplified case and developing to the general case in subsection 3.2.

3.1 The Linear Case

First we introduce the mechanism of propagation of requirements in the linear case, corresponding of a single-flow system containing a sequence of *simple* components.

In this case, each component has one input port and one output port. The requirements associated with the input port address components that are before the current one on the flow. These requirements are *upward requirements*. The requirements associated with the output port address components that are below the current component and we name them *downward requirements*. By default, it is sufficient that a required property associated with a port is provided by a component that is present somewhere on the flow connected to that port. The requirements of a component are not necessarily met by immediate neighbors of that component, but by some components situated further on the corresponding flows. One can specify immediate requirements, which apply only to the next component on that flow. Also negative requirements (a property should not be present in a flow) are possible.

Given a component C, it has, at an arbitrary moment during the composition process, a set CU of n upward requirements, $CU = \{CU_i\}_{i=1...n}$ and a set CD of m downward requirements, $CD = \{CD_i\}_{i=1...m}$. To ensure that the contract of C is fully complied, C must be part of a composition where the components placed above C fulfill all its upward requirements CU and the components below C fulfill all downward requirements CD.

The goal of the composition process is to find the two sets of components that placed above C and below C fulfill all its requirements. Of course, each of these components introduces their own requirements, that have to be also fulfilled.

The initial requirements of C can be propagated to its neighbor components, if the neighbor component does not provide them itself. This mechanism works like delegating the responsibility for these requirements to the neighbor components.

Let a component X provide the set of properties XP and have the upward requirements XU. It makes sense to connect component X above component C (make X the top neighbor of C by connecting the output port of X to the input port of C) if it provides a part of component's C current upward requirements. The subset of C's upward requirements that are fulfilled by component X is a set of properties named $XPCU$,

$$XPCU = XP \bigcap CU \tag{1}$$

If $XPCU$ is not void (that means, component X provides at least some of the upward requirements of component C), component X will be connected at the input port of C.

Most often, component X does not fulfill all the current upward requirements CU of C, such that a subset $XNPCU$ of CU remain not fulfilled:

$$XNPCU = CU - XPCU \tag{2}$$

All the properties belonging to the set $XNPCU$ are requirements that must be fulfilled by other components connected above C. These properties will be added to the set of upward requirements of component X, process that is called *upward propagation of requirements.*

Following this propagation, the new set of upward requirements of component X becomes XU':

$$XU' = XU \bigcup XNPCU \tag{3}$$

The set of properties XU' is a new set of requirements to continue the upward searching of components.

The downward propagation of requirements is defined in a similar manner. Let a component Y provide the set of properties YP and have the downward requirements YD. It makes sense to connect component Y below component C (connect the output port of C to the input port of Y) if it provides a part of component C's current downward requirements. The subset of C's downward requirements that are fulfilled by component Y is a set of properties $YPCD$,

$$YPCD = YP \bigcap CD \tag{4}$$

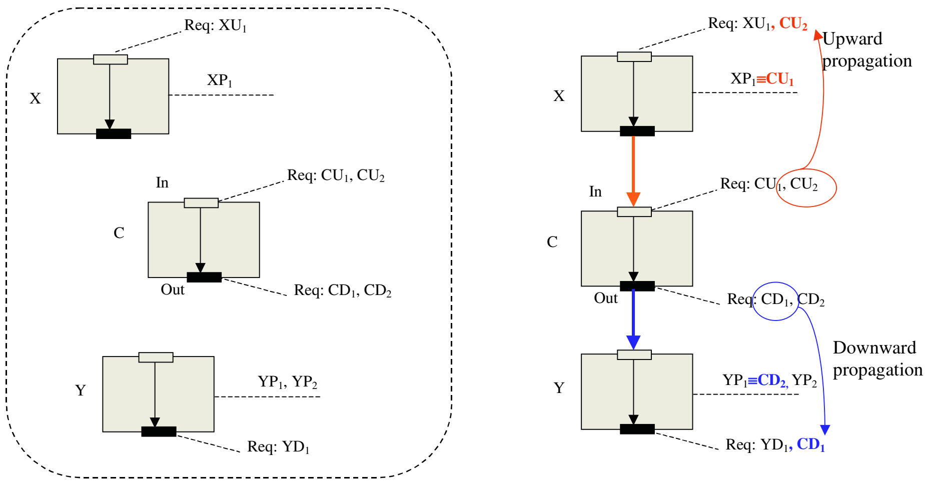
A subset $YNPCD$ of component C's downward requirements remain not fulfilled by component Y:

$$YNPCD = CD - YPCD \tag{5}$$

All the properties belonging to the set $YNPCD$ will be added to the set YD of downward requirements of component Y, process that is called *downward propagation of requirements.*

$$YD' = YD \bigcup YNPCD \tag{6}$$

Figure 1 depicts an example of linear propagation of requirements.



Components in a repository   Use of components in a composition

Fig. 1. Linear propagation of requirements

The example in Figure 1 involves three components: component C, component X, and component Y. Component C has the upward requirements CU, where $CU = \{CU_1, CU_2\}$ and the downward requirements CD, with $CD = \{CD_1, CD_2\}$. A composition that solves the requirements of C must be found. Let a component repository contain among others components X and Y. Component X provides $XP = \{XP_1\}$ and has own upward requirements $XU = \{XU_1\}$. Component Y provides the set of properties $YP = \{YP_1, YP_2\}$ and has own downward requirements $YD = \{YD_1\}$. We ignore at this step of the example the upward requirements of Y and the downward requirements of X.

It is given that property XP_1 matches property CU_1 and property YP_1 matches property CD_2. That means that component X fulfills one of C's upward requirements and component Y fulfills one of C's downward requirements. Component X will be connected on top of component C, making a connection $X.Out \rightarrow C.In$ and component Y will be connected below component C through a connection $C.Out \rightarrow Y.In$. After these connections, the subset $XNPCU$, $XNPCU = \{CU_2\}$, of C's upward requirements remain unfulfilled and will be propagated to the port $X.In$.

The new set of upward requirements of X is now $XU' = \{XU_1, CU_2\}$.

Similarly, the subset of C's downward requirements not fulfilled by Y is $YNPCD = \{CD_1\}$, and will be propagated from C to the port $Y.Out$. The new downward requirements of Y are now $YD' = \{YD_1, CD_1\}$. After having connected component X as the neighbor on top of C and component Y the neighbor below C, the searching for new components continues having XU' and YD' as driving requirements.

The downward requirements of component X as well as the upward requirements of component Y have been ignored until now. If C does not fulfill the downward requirements of X, then a propagation of these from X to C will also occur; the same for upward requirements of Y.

An example of using linear propagation of requirements for the automatic composition of customized network protocol stacks is our early work[1] [ŞMBV03], where a protocol stack is automatically built as a composition of protocol layer components according to client requirements.

The linear case is simple and intuitive, yet not sufficient for the building of more complex systems according to fine-tuned requirements. In the next subsection, the mechanism of propagation of requirements is generalized for components with an arbitrary number of input and output ports that are part of multiflow architectures of hierarchically composable components.

3.2 The General Case

In the general case, the terms "upward" requirements and "downward" requirements become obsolete as they loose their semantics. In this general case, one cannot identify one component as being "over" or "under" another component. This kind of order relationships can be established only between ports that are connected to the same flow. The components have requirements associated with their ports (input ports as well as output ports). The requirements associated with input ports address the flow that comes into this port, while the requirements associated with output ports address the flow that goes out this port.

In order to be able to accurately study the interactions between components with multiple ports that are in a chain of connections, it is necessary to know for each component the relationships between its input port and output ports (the intracomponent pathways as they are named in [SW01]). In our model, the internal flows fixed by the structural constraints of a composable component identify the intracomponent pathways. Propagation of requirements in the case of components with multiple ports will occur only along the intracomponent pathways.

Given a component C, it has NI_C input ports and NO_C output ports. The requirements associated with an output port $C.Out_o$, $o \in [1 \ldots NO_C]$ are a set CO_o of properties.

A component Y has NI_Y input ports and NO_Y output ports and provides the set of properties YP. The component Y fulfills a subset $YPCO_o$ of the requirements CO_o associated with port $C.Out_o$,

$$YPCO_o = YP \bigcap CO_o. \tag{7}$$

If $YPCO_o$ is not empty, the decision to connect port $C.Out_o$ to an input port of component Y (the port $Y.In_i$, $i \in [1 \ldots NI_Y]$) is taken. The selection as connection port of the port i out of the NI_Y input ports is based on additional tests of contracts and is part of the composition strategy, hence not discussed in this section. After doing this connection, most of the cases there will still remain some requirements of Y that are not provided by $C.Out_o$, the set $YNPCO_o$

$$YNPCO_o = CO_o - YPCO_o. \tag{8}$$

[1] Paper written in 2001, delayed in publication.

In the component Y, the input port $Y.In_i$ affects only a subset YFI_i of all the output ports $Yout$ of the component,

$$YFI_i \subset YOut, YOut = \{Y.Out_o | \forall o \in [1 \dots NO_C]\} \quad (9)$$

The elements of YFI_i are those output ports of Y that are situated on intra-component pathways originating in $Y.In_i$. The properties in the set of unfulfilled requirements $YNPCO_o$ will be propagated to all the ports in YFI_i. After the propagation, at every port $Y.Out_o$ the new set of requirements YO'_o will be:

$$\forall Y.Out_o \in YFI_i : YO'_o = YO_o \bigcup YNPCO_o \quad (10)$$

This is the mechanism of propagation of requirements associated with output ports. The propagation of requirements associated with input ports is defined in a similar way.

Figure 2 presents an example of the general case of propagation of requirements.

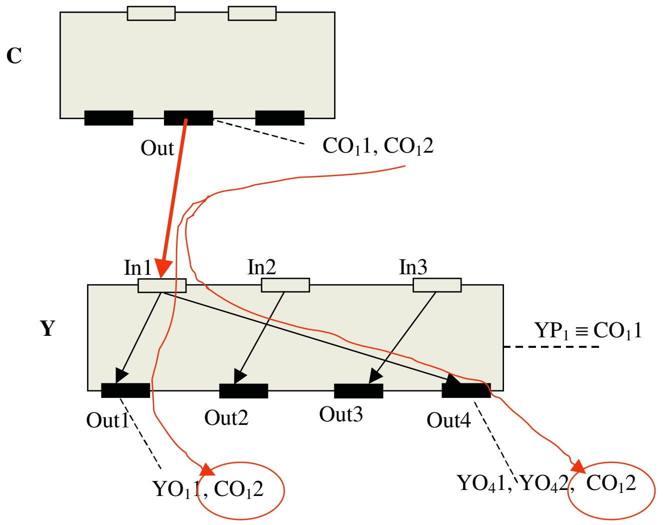


Fig. 2. Propagation of requirements – the general case

The example contains a component C that has one output port $C.Out_1$ with the associated set of requirements $CO_1 = \{CO_11, CO_12\}$.

Component Y in this example has NI_Y=3 inputs and NO_Y=4 outputs. The set of properties provided by Y is $YP = \{YP_1\}$ and it is known that property YP_1 is a match with property CO_11. Component Y has four internal flows and they are: $In_1 \rightarrow Out_1, In_1 \rightarrow Out_4, In_2 \rightarrow Out_2$ and $In_3 \rightarrow Out_3$. If port $C.Out_1$ is connected to port $Y.In_1$, this fulfills requirement CO_11 of port $C.Out_1$. The requirement CO_12 of $C.Out_1$ remains not provided yet and will be propagated to ports $Y.Out_1$ and $Y.Out_4$ (the ports that are connected with input $Y.In_1$).

4 Composition Strategy

We formulate the automatic composition problem as: *given a set of requirements describing the properties of the desired system (the composable target), and a component repository that contains descriptions of available components, the composition process has to find a set of components and their interactions to realize the desired system.* This composition decision occurs through the composition strategy implemented in a *Composer* tool.

We address the requirements driven composition of a multi-flow system by dividing it into subproblems of linear compositions on each flow of the system. Achieving fine-tuned compositions and managing the complexity of the system are possible in our approach by deploying hierarchical composable components. This leads to hierarchically recursive compositions. The driving force of the composition search are the requirements and the propagation of requirements.

First we present the strategy for linear composition on a flow. In the linear case, the composition problem is to determine an ordered sequence of components aligned on a single flow. For presentation terminology, we consider this flow to have a descending orientation. The components align in a layered form on this flow. The client level is the first layer (the "highest" one) and expresses the requirements set REQ imposed for the composable target as its downward requirements. The requirements in REQ are expressed as sets of required properties defined using the same vocabulary as that used for the component descriptions, and possibly as ordering restrictions between properties. Ordering restrictions are generated in most of the cases by the structural constraints. The set of requirements REQ results from the set of requirements CR directly imposed by the client and from the set of requirements SCR that emerge from the structural constraints of the target. Figure 3 depicts the start assumptions for linear composition.

A dummy start component C_0, having REQ as its downward requirements, is created, as Figure 3 shows. The set of requirements REQ is the current driving force for the composition. The search begins looking for components that provide at least a part of the required properties from REQ. If such a component C_x, providing part of REQ, is found, it will be connected below C_0. Component C_x has also its own requirements, upward and downward. The new downward composition driving requirements are now the downward requirements of C_x, together with the propagated part of the initial requirements and the search continues. A component is selected for the solution if it matches at least a subset of the current driving requirements. Similarly, the upward requirements of C_x become the upward composition driving set.

A solution is considered complete when the current composition driving requirements set becomes empty. It is possible that for certain sets of requirements no exact solution can be found. The *Composer* can be configured to respond to this problem in alternative ways, either to relax the client requirements and produce a solution, or to abort composition.

The general case addresses complex systems with multi-flow architecture. An example of how composition results through stepwise refinements is depicted in

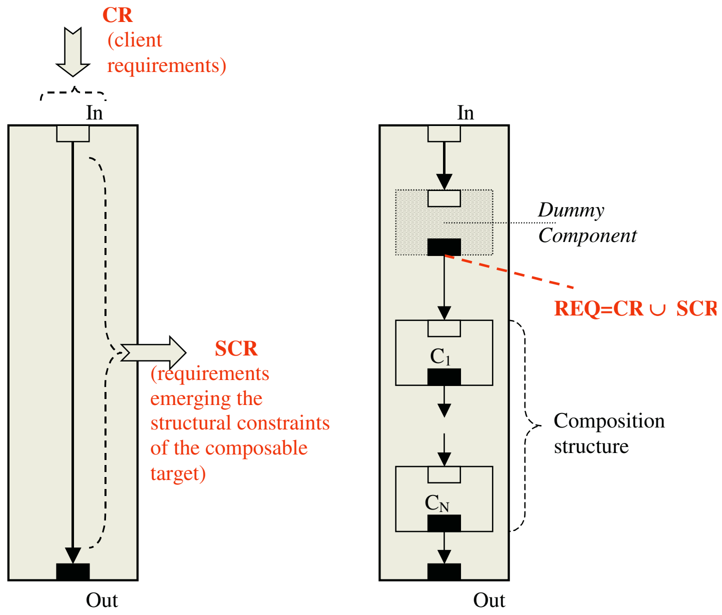


Fig. 3. Composition strategy - the linear case

Figure 4, where the composition target is the internal structure of the composable component C. The set REQ of requirements for the target results by uniting the direct client requirements and the own structural constraints of C.

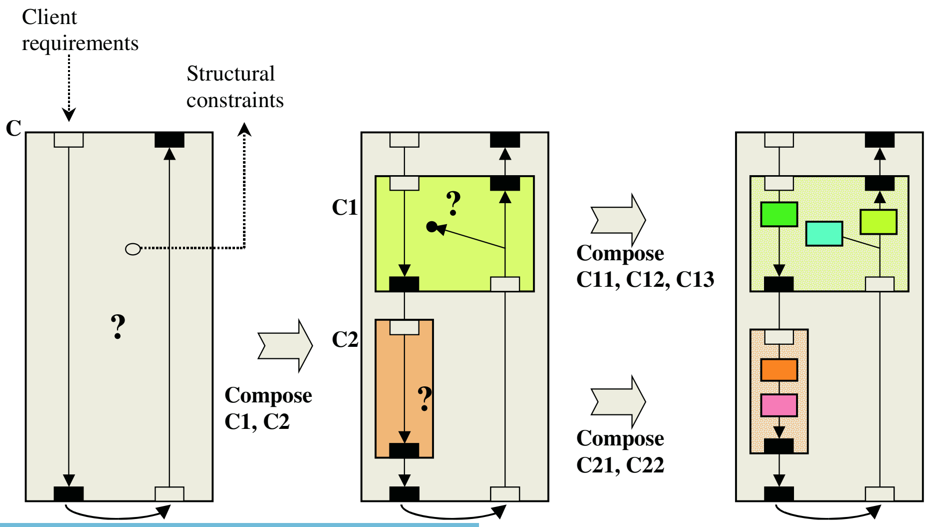


Fig. 4. Composition through stepwise refinements

After a composition search has determined that it wants certain component types ($C1$ and $C2$ in the example in Figure 4) in place to fill in the structure, a new search may be launched for composing the internal structure of these components. Such hierarchically recursive compositions will occur especially if it is necessary to satisfy subproperties of the required properties. Let the original requirements set containe a property $p1$ with the subproperties $p11$ and $p12$ and component $C1$ provide property $p1$. Component $C1$, found to provide $p1$, will have to be fine-tuned so that its internal structure is compliant to the set of subrequirements $p11$, $p12$. The set of required properties $p11$, $p12$ represent direct client requirements for the composition of target $C1$. Together with the structural constraints for $C1$, these requirements lead to the composition of the internal structure of $C1$ from components $C11$, $C12$, $C13$.

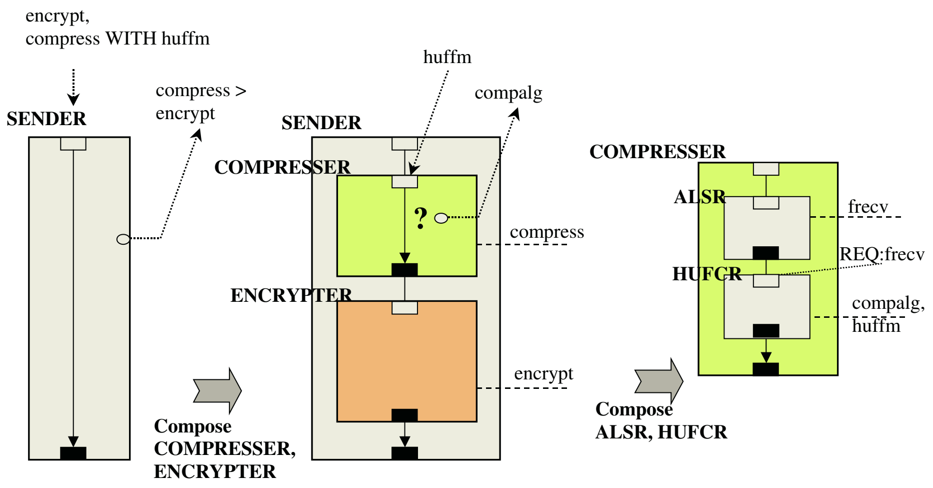


Fig. 5. Composition example

As an example illustrating the concepts presented, we consider the simple scenario depicted in figure 5. The compositional target is a $SENDER$ component that can be customized according to different client requirements. The client requirements in this example contain both encryption and compression of sent data with a particular compression algorithm. These requirements are expressed as properties from the universal properties vocabulary, *encrypt* and *compress*. Property *compress* is specified with a subproperty $huffm$ (compression with the Huffmann algorithm is required). The structural constraints of the $SENDER$ in this example state the ordering restriction that if the property *compress* is present, it should be on the internal flow above property *encrypt*. Through propagation of all these requirements results immediately the solution of composing $SENDER$ from the components $COMPRESSER$ and $ENCRYPTER$ as in the figure. (Without the ordering restriction, both sequences $COMPRESSER$ before $ENCRYPTER$ and $ENCRYPTER$ before

$COMPRESSER$ are possible.) Further, the component $COMPRESSER$ is also a composable one and its internal structure must be composed according to the current client requirements, specified by property $huffm$. The structural constraints of the $COMPRESSER$ component specify that it must contain the property *compalg* on its internal flow (it must contain the implementation of an arbitrary compression algorithm). Starting from these requirements the solution search on the internal flow of $COMPRESSER$ begins. Component $HUFCR$ provides properties $huffm$ and *compalg* and will be deployed inside $COMPRESSER$. $HUFCR$ requires property $frecv$ at its input port (as Huffmann compression requires a static analysis of the data), thus the search continues in the upward direction of the flow and adds the analyzer component $ALSR$ which provides $frecv$. Since no requirements are left unfulfilled, component $COMPRESSER$ is ready composed from components $ALSR$ and $HUFCR$.

5 Discussion and Related Work

A distinctive characteristic of our composition approach is that it works with abstractions of the architectural level. This is according to our insight that architectural style dependent composition models, independent from the application domains, are needed. This permits a generic solution, avoiding coding of specific solutions for each application domain. The approach is to build a system by assuming a certain defined architectural style. Treating component composition in the context of the software architecture is a largely accepted approach ([Ham02], [Wil03], [IT03],[BG97], [KI00], as it makes the problem manageable and eliminates the problems of architectural mismatch. Also, we argue that at the architectural level compositional decisions are made with knowledge of the architectural style, but ignoring technological details of the underlying component model, as long as this provides the infrastructural support needed for runtime assembly of components. Components are described through their properties, seen as facts known about them – in a way similar to Shaw's credentials ([Sha96]). The composition strategy interprets properties in a semantic-unaware way by having a general matching criteria, thus no application-domain specific code occurs in the *Composer*. The general composition strategy described in this paper emerges from our experience with automating composition in two different application domains where systems have multi-flow architectures – self customizable network protocols ([ŞMBV03], [ŞVB02]) and an intelligent environment for virtual instrumentation in measurements and control.

The mechanism of propagation of requirements used in our approach is a generalization rooted in Perry's mechanism of propagation introduced in [Per87], [Per89]. Perry defines a semantic interconnection model for the verification of program semantics, at the level of procedural programming. It extends Hoare's specification of program semantics with pre- and postconditions, proposing another category of clauses, the obligations. Preconditions must be satisfied by the postcondition of an operation that follows on the control flow. Obligations are conditions that must be satisfied by postconditions of operations that precede them on the control flow. In Perry's mechanism, preconditions and obligations are propagated to the interface of the containing module. Our upward requirements may be similar to preconditions, obligations to downward requirements and postconditions to provided properties.

Perry's model deals with the composition of small-grained entities: procedures and functions. Batory et. al. ([BG97], [BCRW00]) propose a similar model for the composition of components in *GenVoca* architectures (layered systems). The entities subject to composition are components, implemented as classes, and are used in layered compositions that can be seen as components being put on top of each other. From the compositional point of view, these components can distinguish only between two interaction points, one upper and one lower interaction point. A particularity of their approach is that a layer provides different properties for the layer on top of it as it provides for the layer below it. This leads to two kinds of pre- and postconditions. Postconditions are named the properties that are provided to the components below it and postrestrictions the properties provided to the components on top. Preconditions are requirements that are directed toward components on top while prerestrictions are requirements directed to components below. In [BG97] an algorithm for the verification of the correctness is given, verification done by downward propagation of postconditions and upward propagation of postrestrictions.

Our approach brings two important contributions. First, we generalize the principle of propagation to non-linear structures. Also we adapt it in the context of components. Our model considers that the provided clause is associated to the component as a whole; a component provides the same properties to all its interacting entities. Requirements are associated with individual ports of the component.

Second, the goal of our model is to serve the automatic component composition (to *generate* the structure of the target assembly) rather than only the verification of a given assembly structure as in the related works. The mechanism of propagation of requirements is the driving force of our searching algorithm. Therefore, in our model the propagated elements are the required properties and not the provided properties.

Our current implementation of a composition algorithm does exhaustive searches and thus has the disadvantage of exponential time. We foresee to implement improvements of it using a search based on heuristics. The mechanism of propagation of requirements as described in this paper will remain a central element of the search.

Our work tackles composition decisions at the semantic level. Other research in automating the composition or adaptation of components deal with the problem at the behavioral level. Different kinds of finite automata or *message sequence charts* (MSC) are used to model the behavior of components ([SR00], [SVSJ03], [IT03]). The behavioral compatibility tests for components check the matching according to syntactic and synchronization criteria. Without considering also semantic level information it is possible that a behavioral test declares as compatible semantical different components that happen to have compatible automata, so semantic checking is needed together with behavioral checking.

6 Conclusions

Our research defines a compositional model for multi-flow architectures that comprises

- a scheme and a language for the description of composable components by semantic–unaware properties and structural constraints.
- a requirements driven composition strategy capable to implement automatic composition decisions starting from the descriptions of the available components and from the requirements for the compositional target.

In this paper, we presented the principles of our composition strategy, introducing the mechanism of propagation of requirements as the central element of our composition strategy. This strategy is implemented by an automatic *Composer* tool that facilitates the building of self-customizable systems. The strengths of our strategy are its simplicity, its application domain independence, and the possibility to compose unanticipated configurations.

References

AG97. Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

BCRW00. Don Batory, Gang Chen, Eric Robertson, and Tao Wang. Design wizards and visual programming environments for GenVoca generators. *IEEE Transactions on Software Engineering*, 26(5), May 2000.

BG97. Don Batory and Bart Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering*, 23(2), February 1997.

Ham02. Dieter K. Hammer. Component-based architecting for component-based systems. In Mehmet Askit, editor, *Software Architectures and Component Technology*. Kluwer, 2002.

IT03. Paola Inverardi and Massimo Tivoli. Deadlock-free software architectures for COM/DCOM applications. *Journal of Systems and Software, Special Issue on Component-Based Software Engineering*, 65(3):173–183, March 2003.

KI00. Christos Kloukinas and Valerie Issarny. Automating the composition of middleware configurations. In *Automated Software Engineering*, pages 241–244, 2000.

Per87. Dewayne E. Perry. Software interconnection models. In *Proceedings of the 9th International Conference of Software Engineering*, pages 61–69, Monterey CA, USA, May 1987.

Per89. Dewayne E. Perry. The logic of propagation in the Inscape environment. In *Proceedings of SIGSOFT '89: Testing, Analysis and Verification Symposium*, Key West FL, USA, December 1989.

Sha96. Mary Shaw. Truth vs knowledge: The difference between what a component does and what we know it does. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 181–185, 1996.

SR00. Heinz Schmidt and Ralf Reussner. Automatic component adaptation by concurrent state machine retrofitting. Technical Report 2000/81, School of Computer Science and Software Engineering, Monash University, Melbourne, Australia, 2000.

SVSJ03. Pieter Schollaert, Wim Vanderperren, Davy Suvee, and Viviane Jonckers. Online reconfiguration of component-based applications in PacoSuite. In *Proceedings of Workshop on Software Composition, affiliated with ETAPS 2003*, volume 82 of *Electronic Notes in Theorethical Computer Science*, Warsaw, Poland, 2003. Elsevier.

SW01. J. A. Stafford and A. L. Wolf. Architecture-level dependence analysis for software systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(4):431–452, August 2001.

Szy97. Clemens Szypersky. *Component Software: Beyond Object Oriented Programming*. Addison-Wesley, 1997.

ŞMBV03. Ioana Şora, Frank Matthijs, Yolande Berbers, and Pierre Verbaeten. Automatic composition of systems from components with anonymous dependencies. In Theo D'Hondt, editor, *Technology of Object-Oriented Languages, Systems and Architectures*, pages 154–169. Kluwer Academic Publishers, 2003.

ŞVB02. Ioana Şora, Pierre Verbaeten, and Yolande Berbers. Using component composition for self-customizable systems. In I. Crnkovic, J. Stafford, and S. Larsson, editors, *Proceedings - Workshop On Component-Based Software Engineering at IEEE-ECBS 2002*, pages 23–26, Lund, Sweden, 2002.

ŞVB03. Ioana Şora, Pierre Verbaeten, and Yolande Berbers. A description language for composable components. In Mauro Pezze, editor, *Fundamental Approaches to Software Engineering, 6th International Conference, Proceedings*, number 2621 in Lecture Notes in Computer Science, pages 22–36. Springer Verlag, 2003.

Wil03. David Wile. Revealing component properties through architectural styles. *Journal of Systems and Software, Special Issue on Component-Based Software Engineering*, 65(3):209–214, March 2003.

Author Index