Lecture Notes in Computer Science 2029

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

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

Heinrich Hussmann (Ed.)

Fundamental Approaches to Software Engineering

4th International Conference, FASE 2001
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2001
Genova, Italy, April 2-6, 2001
Proceedings

Springer

Volume Editor

Heinrich Hussmann
Technische Universität Dresden
Institut für Software- und Multimediatechnik
01062 Dresden, Germany
E-mail:hussmannh@acm.org

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Fundamental approaches to software engineering : 4th international conference ; proceedings / FASE 2001, held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Berlin, Genova, Italy, April 2 - 6, 2001. Heinrich Hussmann (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2029)
ISBN 3-540-41863-6

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

ISSN 0302-9743
ISBN 3-540-41863-6 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Typesetting: Camera-ready by author, PTP Berlin, Stefan Sossna
Printed on acid-free paper SPIN 10782450 06/3142 5 4 3 2 1 0

Foreword

ETAPS 2001 is the fourth 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 comprises five conferences (FOSSACS, FASE, ESOP, CC, TACAS), ten satellite workshops (CMCS, ETI Day, JOSES, LDTA, MMAABS, PFM, RelMiS, UNIGRA, WADT, WTUML), seven invited lectures, a debate, and ten tutorials.

The events that comprise ETAPS address various aspects of the system development process, including specification, design, implementation, analysis and improvement. The languages, methodologies and tools which support these activities are all well within its scope. Different blends of theory and practice are represented, with an inclination towards theory with a practical motivation on 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 programme 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 2001 is hosted by the Dipartimento di Informatica e Scienze dell'Informazione (DISI) of the Università di Genova and has been organized by the following team:

Egidio Astesiano (General Chair)
Eugenio Moggi (Organization Chair)
Maura Cerioli (Satellite Events Chair)
Gianna Reggio (Publicity Chair)
Davide Ancona
Giorgio Delzanno
Maurizio Martelli

with the assistance of Convention Bureau Genova. Tutorials were organized by Bernhard Rumpe (TU München). Overall planning for ETAPS conferences is the responsibility of the ETAPS Steering Committee, whose current membership is:

Egidio Astesiano (Genova), Ed Brinksma (Enschede), Pierpaolo Degano (Pisa), Hartmut Ehrig (Berlin), José Fiadeiro (Lisbon), Marie-Claude Gaudel (Paris), Susanne Graf (Grenoble), Furio Honsell (Udine), Nigel

Horspool (Victoria), Heinrich Hussmann (Dresden), Paul Klint (Amsterdam), Daniel Le Métayer (Rennes), Tom Maibaum (London), Tiziana Margaria (Dortmund), Ugo Montanari (Pisa), Mogens Nielsen (Aarhus), Hanne Riis Nielson (Aarhus), Fernando Orejas (Barcelona), Andreas Podelski (Saarbrücken), David Sands (Göteborg), Don Sannella (Edinburgh), Perdita Stevens (Edinburgh), Jerzy Tiuryn (Warsaw), David Watt (Glasgow), Herbert Weber (Berlin), Reinhard Wilhelm (Saarbrücken)

ETAPS 2001 is organized in cooperation with

the Association for Computing Machinery
the European Association for Programming Languages and Systems
the European Association of Software Science and Technology
the European Association for Theoretical Computer Science

and has received generous sponsorship from:

ELSAG
Fondazione Cassa di Risparmio di Genova e Imperia
INDAM - Gruppo Nazionale per l'Informatica Matematica (GNIM)
Marconi
Microsoft Research
Telecom Italia
TXT e-solutions
Università di Genova

I would like to express my sincere gratitude to all of these people and organizations, the programme 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.

Edinburgh, January 2001

Donald Sannella
ETAPS Steering Committee chairman

Preface

The Conference on Fundamental Approaches to Software Engineering (FASE), as its name indicates, is a pure software engineering conference. However, being part of the ETAPS event, it has a particular profile. It focuses on the application of theoretically founded techniques in practical software engineering and on approaches aiming towards a proper theory of software engineering. In the past, FASE was sometimes mistaken for a Formal Methods conference. However, FASE covers Formal Methods as just a small part of its profile, and even then it only covers application-oriented work on Formal Methods.

As the chairman of the program committee for FASE 2001, I am very happy that this instance of FASE fully coincides with this intended profile of the conference. I am also happy that FASE is an increasingly popular event, as can be seen from the increasing number of submissions. FASE 2001 attracted a record number of 74 submissions. The scope of these submissions was very broad, covering many different areas of software engineering. The program committee had a difficult task in selecting just 22 papers out of the submissions. I am grateful to my colleagues in the program committee that this process went smoothly and lead to a well-balanced program of very good scientific quality. The members of the FASE 2001 program committee were:

Egidio Astesiano (Università di Genova, Italy)
Michel Bidoit (ENS Cachan, France)
Dan Craigen (ORA Ottawa, Canada)
José Fiadeiro (Universidade de Lisboa, Portugal)
Carlo Ghezzi (Politecnico di Milano, Italy)
Heinrich Hussmann (Technische Universität Dresden, Germany)
Cliff Jones (University of Newcastle, UK)
Tom Maibaum (King's College London, UK)
Bernhard Rumpe (Technische Universität München, Germany)
Doug Smith (Kestrel Institute, USA)
Martin Wirsing (Universität München, Germany)

When comparing the program with earlier FASE programs, it is obvious that the section on Formal Methods has decreased in size, but still keeps a prominent position, and puts strong emphasis on practical aspects, like real-world case studies. Some other software engineering topics, such as component-based development, distributed systems, and testing, are included. The biggest group of papers deals with a specification and modeling language which was not even touched upon at the first FASE (1998) and just superficially covered at FASE 1999 and FASE 2000. More than two thirds of the papers explicitly deal with the Unified Modeling Language (UML), in particular with its theoretical foundations and possible extensions. Of course, it is quite controversial whether this language is a scientific achievement in itself, since the evolution of UML is clearly driven by industry and much of UML was defined essentially by establishing a compromise between divergent opinions. Nevertheless, the UML seems to have established itself as one of the major transmission mechanisms between scientific

research and practical application. It is a big step forward that nowadays many fundamental research activities use the UML as a basis and therefore make their results easily accessible for practioners who are knowledgeable of UML. Therefore, I am also very happy with the high percentage of UML-related papers and hope that FASE (and ETAPS in general) will establish itself as a forum for those people who are interested in a seriously scientific approach to UML.

It is also not just by coincidence that our invited speaker for FASE, Bran Selic, comes from a company which is closely related to the invention of UML. His talk, which is summarized in this volume by a short abstract, points out an important challenge to software engineering, that is the integration of physical and quantitative aspects, besides the purely functional view which prevails today.

A scientific event like FASE is always the result of the co-operation of a large group of people. Therefore, I would like to thank the members of the program committee and the additional referees, as listed below, for the enormous amount of work they invested in the selection process. Special thanks go to Ansgar Konermann for his reliable support, in particular by providing and maintaining the Web site on which the virtual program committee meeting was carried out. Many thanks also to Don Sannella, Egidio Astesiano, and the whole ETAPS 2001 team for providing the well-organized framework for this conference.

January 2001

Heinrich Hussmann
FASE 2001 Program Committee chairman

Referees

R. Amadio
L. Andrade
H. Baumeister
M. Becker
D. Bert
B. Blanc
M. Cerioli
D. Clark
T. Clark
P. R. D'Argenio
A. De Lucia
B. Demuth
Th. Dimitrakos
C. Duarte
L. Errington
M. Fischer
S. Fitzpatrick
C. Fox
F. Fünfstück
J.-M. Geib
J. Goubault-Larrecq
A. Haeberer
R. Hennicker
S. Kent
A. Knapp
P. Kosiuczenko
S. Kromodimoeljo
F.-U. Kumichel
K. Lano
A. Lopes
P. Magillo
B. Marre
S. Merz
B. Möller
T. Nipkow
I. Nunes
L. Petrucci
A. Pretschner
G. Reggio
B. Reus
J.-C. Reynaud
M. Saaltink
R. Sandner
Ph. Schnoebelen
J. Thierry
A. Thomas
V. Vasconcelos
F. Voisin
M. Wermelinger
S. Westfold
G. Wimmel
J. Zappe
E. Zucca

Table of Contents

Invited Paper

Physical Programming: Beyond Mere Logic (Invited Talk) 1
Bran Selic (Rational Software Inc., Canada)

Metamodelling

Metamodelling and Conformance Checking with PVS 2
Richard F. Paige, Jonathan S. Ostroff (York University, Toronto)

The Metamodelling Language Calculus: Foundation Semantics for UML . . 17
Tony Clark (King's College London), Andy Evans (University of York), Stuart Kent (University of Kent at Canterbury)

Distributed Components

Compositional Checking of Communication among Observers 32
Ralf Pinger, Hans-Dieter Ehrich (Technische Universität Braunschweig)

Combining Independent Specifications 45
Joy N. Reed (Oxford Brookes University), Jane E. Sinclair (University of Warwick)

Proving Deadlock Freedom in Component-Based Programming 60
Paola Inverardi (Università dell' Aquila), Sebastian Uchitel (Imperial College)

UML

A Real-Time Execution Semantics for UML Activity Diagrams 76
Rik Eshuis, Roel Wieringa (University of Twente)

A CSP View on UML-RT Structure Diagrams 91
Clemens Fischer, Ernst-Rüdiger Olderog, Heike Wehrheim (Universität Oldenburg)

Strengthening UML Collaboration Diagrams by State Transformations . . . 109
Reiko Heckel, Stefan Sauer (University of Paderborn)

Specification of Mixed Systems in KORRIGAN with the Support of a UML-Inspired Graphical Notation 124
Christine Choppy (Université Paris XIII), Pascal Poizat, Jean-Claude Royer (Université de Nantes)

On Use Cases and Their Relationships in the Unified Modelling-Language . 140
Perdita Stevens (University of Edinburgh)

On the Importance of Inter-scenario Relationships in Hierarchical State Machine Design 156
Francis Bordeleau, Jean-Pierre Corriveau (Carleton University, Ottawa)

Towards a Rigorous Semantics of UML Supporting Its Multiview Approach 171
Gianna Reggio, Maura Cerioli, Egidio Astesiano (Università di Genova)

Towards Development of Secure Systems Using UMLsec 187
Jan Jürjens (University of Oxford)

Testing

Grammar Testing 201
Ralf Lämmel (CWI Amsterdam)

Debugging via Run-Time Type Checking 217
Alexey Loginov, Suan Hsi Yong, Susan Horwitz, Thomas Reps (University of Wisconsin-Madison)

Library-Based Design and Consistency Checking of System-Level Industrial Test Cases 233
Oliver Niese (METAFrame Technologies, Dortmund), Bernhard Steffen (University of Dortmund), Tiziana Margaria (METAFrame Technologies, Dortmund), Andreas Hagerer (METAFrame Technologies, Dortmund), Georg Brune, Hans-Dieter Ide (Siemens, Witten)

Demonstration of an Automated Integrated Testing Environment for CTI Systems 249
Oliver Niese, Markus Nagelmann, Andreas Hagerer (METAFrame Technologies, Dortmund), Klaus Kolodziejczyk-Strunck (HeraKom, Essen), Werner Goerigk, Andrei Erochok, Bernhard Hammelmann (Siemens, Witten)

Formal Methods

Semantics of Architectural Specifications in CASL 253
Lutz Schröder, Till Mossakowski (Bremen University), Andrzej Tarlecki (Warsaw University), Bartek Klin (BRICS, Aarhus), Piotr Hoffman (Warsaw University)

Extending Development Graphs with Hiding 269
Till Mossakowski (University of Bremen), Serge Autexier (Saarland University), Dieter Hutter (DKFI, Saarbrücken)

A Logic for the Java Modeling Language JML 284
Bart Jacobs, Erik Poll (University Nijmegen)

A Hoare Calculus for Verifying Java Realizations of OCL-Constrained Design Models 300
Bernhard Reus (University of Sussex at Brighton), Martin Wirsing, Rolf Hennicker (Ludwig-Maximilians-Universität München)

Case Studies

A Formal Object-Oriented Analysis for Software Reliability: Design for Verification 318
Natasha Sharygina (Bell Laboratories), James C. Browne (University of Texas at Austin), Robert P. Kurshan (Bell Laboratories)

Specification and Analysis of the AER/NCA Active Network Protocol Suite in Real-Time Maude 333
Peter C. Ölveczky (SRI, Menlo Park and University of Oslo), Mark Keaton (Litton-TASC, Reading), José Meseguer (SRI, Menlo Park), Carolyn Talcott (Stanford University), Steve Zabele (Litton-TASC, Reading)

Author Index **349**

Physical Programming: Beyond Mere Logic
(Invited Talk)

Bran Selic

Rational Software Inc., Canada
bselic@rational.com

Abstract. Plato believed in a "pure" reality, where ideas existed in their perfection into eternity. What we perceive as reality, he claimed, is merely a flawed shadow of this ideal world. Many mathematicians find this view appealing since it is precisely this universe of ideas that is the subject of their exploration and discovery. The computer, and more specifically, software, seem perfectly suited to this viewpoint. They allow us to create our own reality, one in which we can simply ignore the underlying physics, forget the tyranny of inertial mass, the drudgery of dealing with machinery that leaks or parts that do not quite fit. But, can we? Even in the ideal world with infinite resources, we have discovered that there are limits to computability. However, the situation with computers and software is much more dire than mere limits on what can be computed. As computers today play an indispensable part of our daily lives we find that more and more of the software in them needs to interact with the physical world. Unfortunately, the current generation of software technologies and practices are constructed around the old Platonic ideal. Standard wisdom in designing software encourages us to ignore the underlying technological platform Ð after all, it is likely to change in a few years anyway Ð and focus exclusively on the program "logic". However, when physical distribution enters the picture, we find that mundane things such as transmission delays or component failures may have a major impact on that logic. The need to deal with this kind of raw physical "stuff" out of which the software is constructed has been relegated to highly specialised areas, such as real-time programming. The result is that we are singularly unprepared for the coming new generation of Internet-based software. Even languages that were nominally designed for this environment, such as Java, are lacking in this regard. For example, it has no facility to formally express that a communication between two remote parts must be performed within a specified time interval. In this talk, we first justify the need to account for the physical aspects when doing software design. We then describe a conceptual framework that allows us to formally specify and reason about such aspects. In particular, it requires that we significantly expand the concept of *type* as currently defined in software theory.

References

1. B. Selic, "A generic framework for modeling resources with UML", *IEEE Computer*, June 2000.

H. Hussmann (Ed.): FASE 2001, LNCS 2029, p. 1, 2001.

Metamodelling and Conformance Checking with PVS

Richard F. Paige and Jonathan S. Ostroff

Department of Computer Science, York University,
Toronto, Ontario M3J 1P3, Canada. {paige,jonathan}@cs.yorku.ca

Abstract. A metamodel expresses the syntactic well-formedness constraints that all models written using the notation of a modelling language must obey. We formally capture the metamodel for an industrial-strength object-oriented modelling language, BON, using the PVS specification language. We discuss how the PVS system helped in debugging the metamodel, and show how to use the PVS theorem prover for conformance checking of models against the metamodel. We consider some of the benefits of using PVS's specification language, and discuss some lessons learned about formally specifying metamodels.

1 Introduction

Modelling languages such as UML [2], BON [7], and others have been used to capture requirements, describe designs, and to document software systems. Such languages are supported by tools, which aid in the construction of models, the generation of code from models, and the reverse engineering of models from code.

A modelling language consists of two parts: a *notation*, used to write models; and a *metamodel*, which expresses the syntactic well-formedness constraints that all valid models written using the notation must obey [2]. Metamodels serve several purposes that may be of interest to different modelling language users.

- **Modellers:** metamodels should be easy to understand by modellers. Thus, metamodels should be expressed so that their fundamental details can be easily explained to modellers, without requiring them to understand much formalism.
- **Tool Builders:** metamodels provide specifications for tool builders who are constructing applications to support the modelling language. Thus, metamodels should be precise and not open to misinterpretation.
- **Modelling Language Designers:** language designers have the responsibility to ensure that metamodels are consistent. Thus, metamodels should be expressed in a formalism so that automated reasoning about it can be carried out.

Different modelling language users have different goals, and therefore a metamodel must possess a collection of different characteristics. Metamodels must be understandable, to assist in the use of the language and its supporting tools. They should be unambiguous and not open to misinterpretation. A metamodel should be expressed in a form amenable to tool-based analysis, e.g., for consistency checking. And, to best deal with complexity and issues of scale, a metamodel should be well-structured.

In this paper, we present a formal specification of the metamodel of the BON object-oriented (OO) modelling language [7], written using the PVS specification language [3].

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 2–16, 2001.

The PVS language has been designed for automated analysis, using the PVS system, which provides a theorem prover and typechecker. We construct the formal specification in two steps. We first specify the BON metamodel informally, using BON itself. In this manner, we use BON's structuring facilities to help understand the key abstractions used in the metamodel and their constraints, before writing a formal specification. From the BON version of the metamodel, we then construct a set of PVS theories that capture the metamodel. The PVS theories can then be used, in conjunction with the PVS system, to analyze and answer questions about the metamodel.

The specific contributions of this paper are as follows.

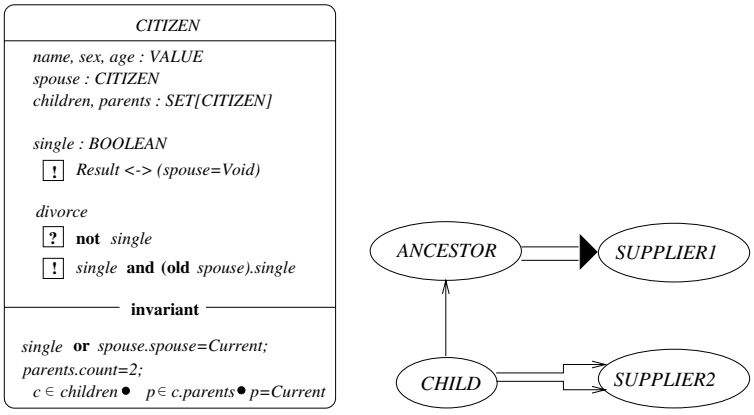
1. A formal specification of the syntactic well-formedness constraints for BON in the PVS language. To our knowledge, this is the first formal specification of the full metamodel of an OO modelling language in a form that is also amenable to automated analysis.
2. A description of how the PVS language can be used for metamodelling, and how the PVS system can be used to help debug and verify the metamodel.
3. Examples of how the PVS system can be used to reason about the metamodel. As a specific example, we show how to prove that BON models satisfy the metamodel.

2 An Overview of BON

BON is an OO method possessing a recommended process as well as a graphical language for specifying object-oriented systems. The fundamental construct in the BON language is the class. A class has a name, an optional class invariant, and zero or more features. A feature may be a *query* – which returns a value and does not change the system state – or a *command*, which does change system state but returns nothing. Fig. 1(a) contains an example of a BON model for the interface of a class *CITIZEN*. Features are in the middle section of the diagram (there may be an arbitrary number of feature sections, each annotated with names of classes that may access the features). Features may optionally have contracts, written in the BON assertion language, in a pre- and postcondition form. In postconditions, the keyword **old** can be used to refer to the value of an expression when the feature was called. Similarly, the implicitly declared variable *Result* can be used to constrain the value returned by a query. An optional class invariant is at the bottom of the diagram. The class invariant is an assertion that must be *true* whenever an instance of the class is used by another object. In the invariant, *Current* refers to the current object.

Classes and features can be tagged with a limited set of stereotypes. The root class contains a feature from which computation begins. A class name with a $*$ next to it indicates the class is deferred, i.e., some features are unimplemented and thus the class cannot be instantiated; a + next to a class name indicates the class can be instantiated. A ++ next to a feature name indicates the feature is redefined, and its behaviour is changed from behaviour inherited from a parent.

In Fig. 1(a), class *CITIZEN* has seven queries and one command. For example, *single* is a *BOOLEAN* query while *divorce* is a parameterless command that changes the state of an object. $SET[G]$ is a generic predefined class with generic parameter G. $SET[CITIZEN]$ thus denotes a set of objects each of type *CITIZEN*.



(a) Citizen interface (b) BON relationships

Fig. 1. BON syntax for interfaces and relationships.

BON models usually consist of classes organized in *clusters* (drawn as dashed rounded rectangles – see Section 3), which interact via two kinds of relationships.

- **Inheritance:** Inheritance defines a subtype relationship between child and one or more parents. The inheritance relationship is drawn between classes *CHILD* and *ANCESTOR* in Fig. 1(b), with the arrow directed from the child to the parent class. In this figure, classes have been drawn in their compressed form, as ellipses, with interface details hidden.
- **Client-supplier:** there are two client-supplier relationships, association and aggregation. Both relationships are directed from a *client* class to a *supplier* class. With association the client class has a reference to an object of the supplier class. With aggregation the client class contains an object of the supplier class. The aggregation relationship is drawn between classes *CHILD* and *SUPPLIER*2 in Fig. 1(b), whereas an association is drawn from *ANCESTOR* to class *SUPPLIER*1.

3 BON Specification of the Metamodel

We now present an informal specification of the BON metamodel, written in BON itself. This description is aimed at promoting an understanding of the fundamental abstractions and relationships that BON models use. We use BON to informally capture the metamodel, and the BON description will be used to help produce a formal specification of the metamodel in the PVS language.

Fig. 2 contains an abstract depiction of the BON metamodel. BON models are instances of the class *MODEL*. Each model has a set of abstractions. The two clusters, representing abstractions and relationships, will be detailed shortly.

Fig. 2. The BON metamodel, abstract architecture.

The class *MODEL* possesses a number of features and invariant clauses that will be used to capture the well-formedness constraints of BON models. These features are depicted in Fig. 3, which shows the interface for *MODEL*. We will not provide details of the individual clauses of the class invariant of *MODEL* (these can be found in [5]).

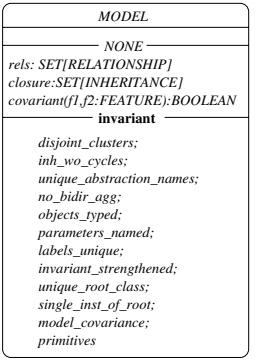


Fig. 3. Interface of class *MODEL*.

3.1 The Relationships Cluster

The relationships cluster describes the four basic BON relationships, as well as constraints on their use. The details of the cluster are shown in Fig. 4.

There are several important things to observe about Fig. 4.

- *Type redefinition:* BON allows types of features to be covariantly redefined [1] when they are inherited: a type in the signature of a feature can be replaced with a subtype. In class *RELATIONSHIP*, the attributes *source* and *target* are given types *ABSTRACTION*. In *INHERITANCE* and *CLIENT_SUPPLIER*, the types are redefined to *STATIC_ABSTRACTION*.
- *Aggregation:* an aggregation relationship cannot target its own source; this is precisely captured by the invariant on *AGGREGATION*. Associations, however, may target their source because they depict reference relationships.
- *Inheritance:* a class cannot inherit from itself. This is captured by the invariant of class *INHERITANCE*.

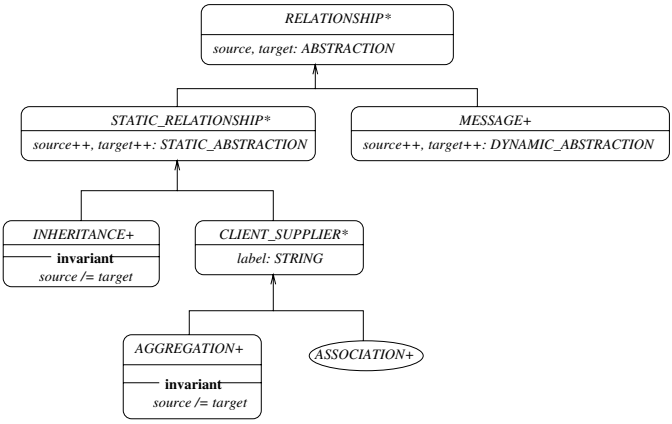


Fig. 4. The relationships cluster.

3.2 The Abstractions Cluster

The abstractions cluster describes the abstractions that may appear in a BON model. It is depicted in Fig. 5. Details of the invariant clauses of individual classes may be found in [5]; PVS specifications of several clauses will be presented in Section 4.

The features cluster of Fig. 5 will be described in Section 3.3. It contains the constraints relevant to features of classes. In particular, this cluster introduces the class *FEATURE* to represent the abstract notion of a feature belonging to a class.

ABSTRACTION is a deferred class: instances of *ABSTRACTION*s cannot be created. Classification is used to separate all abstractions into two subtypes: static and dynamic abstractions. Static abstractions are *CLASS*es and *CLUSTER*s. Dynamic abstractions are *OBJECT*s and *OBJECT_CLUSTER*s. Clusters may contain other abstractions according to their type, i.e., static clusters contain only static abstractions.

3.3 The Features Cluster

The features cluster describes the notion of a feature that is possessed by a class. Features have optional parameters, an optional precondition and postcondition, and an optional frame. The pre- and postcondition are assertions; the cluster that metamodels assertions can be found in [5]. Query calls may appear in assertions; the set of query calls that appear in an assertion must be modelled in order to ensure that the calls are valid according to the export policy of a class. Each feature will thus have a list of *accessors*, which are classes that may use the feature as a client. A call consists of an entity (the target of the call), a feature, and optional arguments. The frame is a set of parameterless queries that the feature may modify. Fig. 6 depicts the cluster.

4 PVS Specification of the Metamodel

In this section, we present a formal specification of the BON metamodel in the PVS specification language. We attempt to parallel the structure of the BON metamodel in

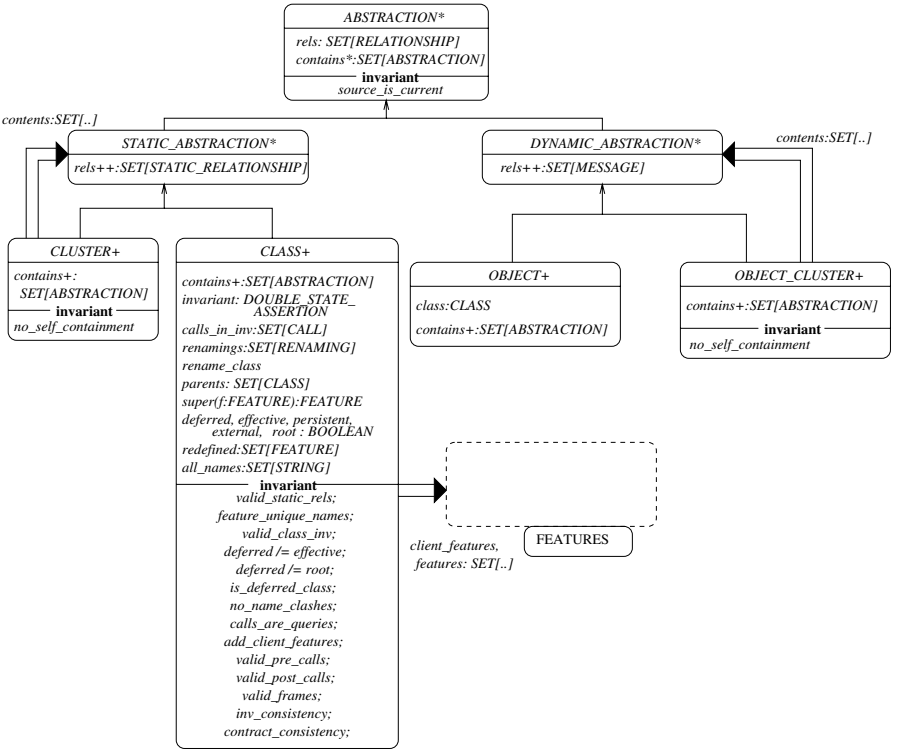


Fig. 5. The abstractions cluster.

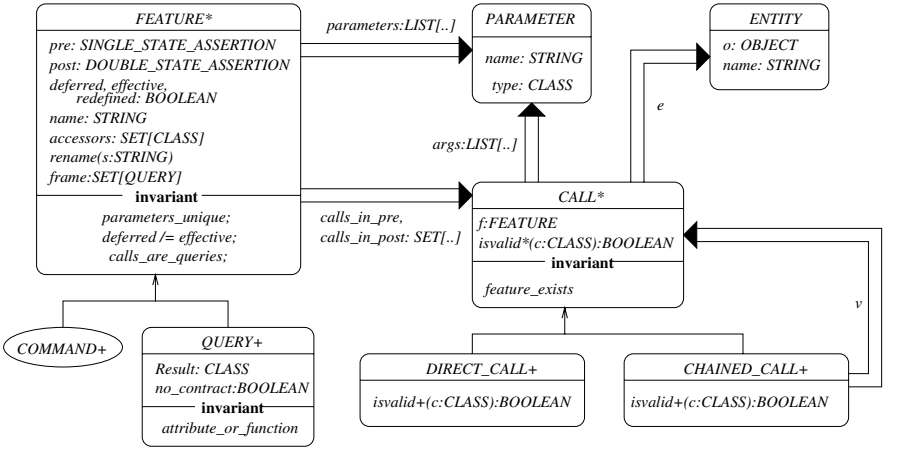


Fig. 6. BON metamodel, features cluster.

the PVS language. We present the PVS version of the metamodel selectively, and attempt to give the flavour of using PVS for this purpose (the full metamodel can be found in [5]).

4.1 Theory of Abstractions

A BON model can contain a number of abstractions, specifically classes, clusters, objects, and clusters of objects. To express this in PVS, we introduce a new non-empty type and a number of subtypes, effectively mimicking the inheritance hierarchy presented in Fig. 5. This information is declared in the PVS theory `abs_names.pvs`.

```
abs_names: THEORY
BEGIN
  ABS: TYPE+

  % Static and dynamic abstractions.
  STATIC_ABS, DYN_ABS: TYPE+ FROM ABS

  % Instantiable abstractions
  CLASS, CLUSTER: TYPE+ FROM STATIC_ABS
  OBJECT, OBJECT_CLUSTER: TYPE+ FROM DYN_ABS
END abs_names
```

The PVS theory `abstractions` then uses `abs_names` to introduce further modelling concepts as well as the constraints on abstractions that appear in models. Further concepts that we need to model include features and entities (that appear in calls).

```
abstractions: THEORY
  IMPORTING abs_names, rel_names

  % Features are queries or commands.
  FEATURE: TYPE+
  QUERY, COMMAND: TYPE+ FROM FEATURE
  ENTITY: TYPE+
```

We can now model feature calls (which may appear in an assertion associated with a feature). Parameters and arguments are modelled as functions from features to finite sequences of records. Calls are either direct (of the form $o.f$) or chained (of the form $o.p.q$).

```
  PARAM: TYPE = [# name:string, param_type: CLASS #]
  IMPORTING sequences[PARAM]
  parameters: [ FEATURE -> finite_sequence[PARAM] ]

  % Direct and chained calls.
  CALL: TYPE+
  DIRECT_CALL, CHAINED_CALL: TYPE+ FROM CALL
```

Primitive BON classes, e.g., *INTEGER*, are modelled as PVS constants: objects of `CLASS` type. We also define conversions so that the type checker can automatically convert BON primitives into PVS types.

```
  bool_object, int_object, string_object, real_object, any_object: CLASS

  % Example conversions that the PVS typechecker can automatically apply.
  interp_bool:   [ bool_object -> bool ]
  interp_int:    [ int_object -> int ]
  CONVERSION interp_bool, interp_int
```

We must now describe constraints on abstractions. In the BON version of the metamodel, these took the form of features and class invariants. In PVS, the well-formedness constraints will appear as functions, predicate subtypes, and axioms. We start by defining a number of functions that will later be used to constrain the model.

```
  % The class that an object belongs to, and the features of a class.
  object_class: [ OBJECT -> CLASS ]
  class_features: [ CLASS -> set[FEATURE] ]

  % The contents of a cluster. Note that clusters may be nested.
  cluster_contents: [ CLUSTER -> set[STATIC_ABS] ]
```

A number of constraints will have to be written on features. To accomplish this, we introduce a number of functions that will let us acquire information about a feature, such as its properties, precondition, and postcondition.

```
  feature_pre, feature_post: [ FEATURE -> bool ]

  % Properties of a feature.
  deferred_feature, effective_feature, redefined_feature: [ FEATURE -> bool ]

  % The set of classes that can legally access a feature.
  accessors: [ FEATURE -> set[CLASS] ]
```

We need to be able to capture the concept of a legal set of calls. Consider an assertion in BON, e.g., a precondition. Such an assertion may call a query if the class owning the query has given permission to do so. To accomplish this, we introduce functions that give us all the calls associated with a precondition, postcondition, and invariant.

```
  calls_in_pre, calls_in_post: [ FEATURE -> set[CALL] ]
  calls_in_inv: [ CLASS -> set[CALL] ]
```

We now provide examples of axioms, which define the constraints on BON models. The first example ensures that all features of a class have unique names (BON does not permit overloading based on feature names or signatures).

```
feature_unique_names: AXIOM
 (FORALL (c:CLASS): (FORALL (f1,f2:FEATURE):
      (member(f1,class_features(c)) AND member(f2,class_features(c)))
      IMPLIES (feature_name(f1) = feature_name(f2)) IMPLIES f1=f2))
```

A further axiom ensures that clusters do not contain themselves.

```
  no_self_containment_cl: AXIOM
    (FORALL (cl:CLUSTER): not member(cl,cluster_contents(cl)))
```

Here is an example of specifying that an assertion is valid according to the export policy used in a model. The axiom `valid_precondition_calls` ensures that: (a) all calls in a precondition are legal (according to the accessor list for each feature); and (b) all calls in the precondition are queries.

```
valid_precondition_calls: AXIOM
  (FORALL (c:CLASS): (FORALL (f:FEATURE): member(f, class_features(c)) IMPLIES
       (FORALL (call:CALL): member(call, calls_in_pre(f)) IMPLIES
         QUERY_pred(f(call)) AND call_isvalid(f(call)))))
```

Classes may possess stereotypes, e.g., they may be deferred or effective. Here is an example, showing that a class cannot be both deferred and effective.

```
deferred_effective_ax: AXIOM
  (FORALL (c:CLASS): (NOT (deferred_class(c) IFF effective_class(c))))
```

4.2 Theory of Relationships

The theory of relationships defines the three basic relationships and the well-formedness constraints that exist in BON. To express the relationships in PVS, we introduce a new non-empty type and a number of subtypes. As with abstractions, we mimic the inheritance hierarchy that was presented in Fig. 4.

```
rel_names: THEORY
BEGIN
  % The abstract concept of a relationship.
  REL: TYPE+

  % Instantiable relationships.
  INH, C_S, MESSAGE: TYPE+ FROM REL
  AGG, ASSOC: TYPE+ FROM C_S
END rel_names
```

The `rel_names` theory is then used by the `relationships` theory. In BON, all relationships are directed (or targetted). Thus, each relationship has a source and a target, and these concepts are modelled using PVS functions.

```
relationships: THEORY
BEGIN
  IMPORTING rel_names, abstractions

  % Examples of the source and target of a relationship.
  inh_source, inh_target: [ INH -> STATIC_ABS ]
  cs_source, cs_target:   [ C_S -> STATIC_ABS ]
```

Now we can express constraints on the functions. We give one example of relationship constraints: that inheritance relationships cannot be self-targetted.

```
% Inheritance relationships cannot be directed from an abstraction to itself.
inh_ax: AXIOM  (FORALL (i:INH): NOT (inh_source(i)=inh_target(i)))
```

The theory of relationships is quite simple, because many of the constraints on the use of relationships are *global* constraints that can only be specified when it is possible to discuss all abstractions in a model. Thus, further relationship constraints will be added in the next section, where we describe constraints on models themselves.

4.3 The Metamodel Theory

The PVS theory `metamodel` uses the two previous theories – of abstractions and relationships – to describe the well-formedness constraints on all BON models. Effectively, the PVS theory `metamodel` (described below) mimics the structure of the BON model in Fig. 2: a model consists of a set of abstractions.

```
metamodel: THEORY
BEGIN
IMPORTING abstractions, relationships

% A BON model consists of a set of abstractions.
abs:  SET[ABS]
rels: SET[REL]
```

Now we must write constraints on how models can be formed from a set of abstractions. The first constraint we write ensures that inheritance hierarchies do not have cycles. We express this by calculating the *inheritance closure*, the set of all inheritance relationships that are either explicitly written in a model, or that arise due to the transitivity of inheritance.

```
inh_closure: SET[INH]

% Closure axiom #1: any inheritance relationship in a model is also
% in the inheritance closure.
closure_ax1: AXIOM
  (FORALL (r:INH): member(r,rels) IMPLIES member(r,inh_closure))

% Closure axiom #2: all inheritance relationships that arise due to
% transitivity must also be in the inheritance closure.
closure_ax2: AXIOM
  (FORALL (r1,r2:INH):
     (member(r1,rels) AND member(r2,rels) AND inh_source(r1)=inh_target(r2))
     IMPLIES (EXISTS (r:INH): member(r,inh_closure) AND
        inh_source(r)=inh_source(r2) AND inh_target(r)=inh_target(r1)))

% Inheritance relationships must not generate cycles.
inh_wo_cycles: AXIOM
  (FORALL (i:INH): member(i,inh_closure) IMPLIES
    NOT (EXISTS (r1:INH): (member(r1,rels) AND i/=r1) IMPLIES
       inh_source(i)=inh_target(r1) AND inh_target(i)=inh_source(r1)))
```

Two further functions will be used in ensuring syntactic covariant redefinition of features. In BON, if a feature's signature is redefined when it is inherited, it can be changed to a subtype.

```
% is_subtype is true iff the second arg. is a descendent of the first
is_subtype: [ CLASS,CLASS -> bool ]

% The function covariant takes two features and results in true
% iff the second feature covariantly redefines the first.
covariant: [ FEATURE,FEATURE -> bool ]

covariant_ax1: AXIOM
  (FORALL (que1,que2:QUERY): covariant(que1,que2) IFF
    length(parameters(que1))=length(parameters(que2)) AND
    (FORALL (i:{j:nat|j<length(parameters(que1))}):
      is_subtype(param_type(parameters(que1)(i)),
                 param_type(parameters(que2)(i))) AND
      is_subtype(query_result(que1),query_result(que2)))
```

The primary purpose of introducing the `covariant` function is to ensure that redefined features obey the syntactic aspects of the covariant rule. The syntactic aspects of covariance are captured in the metamodel via the axiom `model_covariance`, which ensures that all feature redefinitions obey the covariance rule.

```
model_covariance: AXIOM
  (FORALL (c:CLASS): member(c,abst) IMPLIES
    (FORALL (f:FEATURE): member(f,redefined_features(c)) IMPLIES
                         covariant(f,super(c,f)))
```

We write an axiom demonstrating a well-formedness constraint on clusters: all clusters in a model are disjoint or nested.

```
% All clusters in a model are disjoint.
disjoint_clusters: AXIOM
  (FORALL (c1,c2:CLUSTER): (member(c1,abst) AND member(c2,abst)) IMPLIES
  (c1=c2 OR empty?(intersection(cluster_contents(c1),cluster_contents(c2)))))

% No bidirectional aggregation relationships are allowed.
no_bidir_agg: AXIOM
  (NOT (EXISTS (r1,r2:AGG): (member(r1,rels) AND member(r2,rels))
  IMPLIES (cs_source(r1)=cs_target(r2) AND cs_target(r1)=cs_source(r2)))
```

A somewhat complicated axiom to formalize is to ensure that labels appearing on a client-supplier relationship do not clash with names appearing in the feature list of the relationship source, nor with any other client-supplier relationship from the same source. This is reasonably straightforward to formalize in the case where the source of the relationship is a class, but it becomes more complex when the source is a cluster. First we present the case where the source is a class.

```
labels_unique_ax1: AXIOM
  (FORALL (cs:C_S): (member(cs,rels) AND CLASS_pred(cs_source(cs))
    IMPLIES NOT member(cs_label(cs),
      { n:string | (EXISTS (f:FEATURE):
        member(f,class_features(cs_source(cs))) IMPLIES
        n=feature_name(f)) })  AND
        NOT (EXISTS (cs2:C_S): (member(cs2,rels) IMPLIES
         cs_source(cs2)=cs_source(cs) AND cs_label(cs)=cs_label(cs2))) ))
```

A second axiom is needed in the case where the source of the client-supplier relationship is a cluster. In this case, we must require that at least one class contained within the cluster does not have the name appearing on the relationship label.

```
labels_unique_ax2: AXIOM
  (FORALL (cs:C_S): (member(cs,rels) AND CLUSTER_pred(cs_source(cs)))
    IMPLIES (EXISTS (c:CLASS): member(c,all_classes(cs_source(cs))
      IMPLIES NOT member(cs_label(cs),all_names(c))))
```

It was only when typechecking the PVS theories that we discovered the need for `labels_unique_ax2`. Our original formulation considered only the case where the source of a client-supplier relationship is a class. The typechecker provided us with an obligation with the assumption `CLASS_pred(cs_source(cs))`, which is not true for all BON models, since client-supplier relationships may be from clusters as well as classes. Thus, PVS provided us with a counterexample to our original assumptions and thereby suggested extra constraints that needed to be formalized.

The complete metamodel typechecks without any user intervention. It can be found in [5].

5 Conformance Checking with the Metamodel

The metamodel presented in the previous section can be used to check that BON models, which are instances of the metamodel, obey the well-formedness constraints. Conformance checking is by proving PVS `CONJECTURE`s using the axioms of the metamodel. We present two examples to demonstrate the general approach. More examples and further discussion can be found in [5]. The two BON models in Fig. 7 will be used to demonstrate the process.

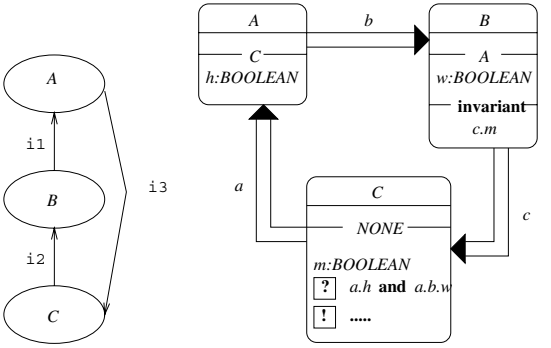


Fig. 7. Models for conformance checking.

5.1 Inheritance Cycles

We start by showing that a model that possesses cycles in its inheritance graph does not satisfy the metamodel. Consider Fig. 7(a) (labels are for reference only); this model is not well-formed because of the cycle-introducing inheritance relationship from class A to class C. If we can describe this model in PVS, then we should be able to conject and prove that it is not well-formed. The conjecture is captured in the following theory.

```
use_metamodel2: THEORY
BEGIN
IMPORTING metamodel
    a,b,c: VAR CLASS
    i1,i2,i3: VAR INH

    no_inh_cycles: CONJECTURE
    (NOT (EXISTS (a,b,c:CLASS): member(a,abst) AND member(b,abst) AND
       member(c,abst) AND a/=b AND b/=c AND c/=a IMPLIES
      (EXISTS (i1,i2,i3:INH): (member(i1,rels) AND member(i2,rels) AND
        member(i3,rels) AND i1/=i2 AND i2/=i3 AND i3/=i1 IMPLIES
        member(i1,static_rels(b) AND member(i2,static_rels(c) AND
        member(i3,static_rels(a)) AND inh_source(i1)=b AND inh_source(i2)=c AND
        inh_source(i3)=a AND inh_target(i1)=a AND inh_target(i2)=b AND
        inh_target(i3)=c))))
END use_metamodel2
```

The conjecture can be explained as follows: there cannot exist a model consisting of the distinct classes a, b, and c with three inheritance relationships $i1$, $i2$, and $i3$ such that $i1$ is directed from b to a, $i2$ is directed from c to b, and $i3$ is directed from a to c. To prove the conjecture with PVS requires use of three axioms, two of which define the inheritance closure of a model, with the third being `inh_wo_cycles`. After instantiating the axioms with the abstractions contained in the model, the conjecture proves automatically using `(grind)`. See [5] for the full proof.

5.2 Obeying Export Policies of Classes

As a second example, we show how to check that a model correctly obeys the export policies of all classes in the model. Consider the BON model in Fig. 7(b). Note that m is a private feature of class C; thus the call $c.m$ in the invariant of B is illegal. Similarly, the call $a.b.w$ in class C is illegal in the precondition of m, because w is accessible only to the client A. We would like to show that this model does not obey the constraints in the BON metamodel. We will show that, as an example, the invariant of B is not well-formed. To prove that the model is not well-formed, we show that the class invariant for B is ill-formed, by conjecting that the model in Fig. 7(b) cannot exist. The full conjecture contains a number of terms that are not relevant to the proof (they can be found in [5]) but which would be included in a completely mechanical derivation of the conjecture; we only include terms relevant to the proof in this presentation, due to space constraints.

```
info_hiding: THEORY
 BEGIN
  IMPORTING metamodel

  a, b, c: VAR CLASS
  h, w, m: VAR QUERY
  call1, call2, call3: VAR CALL

  test_info_hiding: CONJECTURE
    (NOT (EXISTS (a,b c: CLASS): EXISTS (h,w,m:QUERY):
     EXISTS (call1,call2,call3: CALL):
      member(c,accessors(h)) AND member(a,accessors(w)) AND
      empty?(accessors(m)) AND f(call1)=h AND f(call2)=w AND
      f(call3)=m AND member(call1,calls_in_pre(m)) AND
      member(call2,calls_in_pre(m)) AND member(call3,calls_in_inv(b))))
 END info_hiding
```

To prove the conjecture, we first skolemize three times, then flatten. We introduce the axiom `valid_class_invariant`, and substitute class *B* and call *call3* for the bound variables of this axiom. We use `typepred` to bring the type assumptions on *m* into the proof, and then one application of `grind` proves the conjecture automatically. The model is invalid according to the well-formedness constraints of the metamodel.

6 Discussion and Conclusions

By producing a formal metamodel for BON, we have taken a step towards placing the modelling language on a solid mathematical basis. We have captured the well-formedness constraints that all BON models must obey, thus describing core information that is essential for all tool builders and modellers to understand.

We learned several things about PVS and metamodelling in carrying out this exercise. For one, we found the BON version of the metamodel extremely useful in constructing the PVS version. The BON version provided structuring information and indications as to how PVS theories might be related. We also determined several helpful heuristics that can be used, in general, to help model object-oriented concepts in PVS. Class hierarchies can be modelled using PVS types and subtypes. Class features can be described as functions that take a variable as an argument. Commands are PVS functions that take an invoking object as an argument and return a new object. We found the PVS `CONVERSION` facility ideal for transforming BON built-in primitives, e.g., *INTEGER*, into PVS types. Finally, we found that we could model BON's covariant redefinition of feature signatures via PVS's subtyping mechanism. In principle, a formal translation of BON models into PVS, and thereafter a tool, could be developed based on these heuristics.

We found the PVS type checker particularly helpful in debugging the metamodel. Our initial metamodel contained several errors and omissions – e.g., that a client-supplier relationship must always be from a class source, and that we erroneously required that a feature must have one or more parameters – that the checker caught automatically. This was used in updating the metamodel as it was being constructed.

By giving a PVS specification of the metamodel, we have the additional advantage of being able to use the PVS system to analyze the metamodel. The PVS system allowed us to carry out conformance checks of models against the metamodel, as demonstrated in Section 5. The PVS specification allows us to do more than check models against

the metamodel: it allows us to ask *questions* about the metamodel, in particular, about emergent properties of the metamodel. These properties are not explicitly described via the axioms of the metamodel itself; rather, they are logical consequences of the axioms. Thus, the PVS system can be used to help users of the metamodel answer pertinent questions they may have about the metamodel.

Much work remains to be done. We plan to carry out further examples of conformance checking, particularly concentrating on examples that require inductive proofs. We also plan to validate the BON metamodel itself – i.e., prove that the version of the metamodel described in Section 3 is a valid BON model; this will give us greater confidence in the validity of our work. Comparisons of our work with other formal specifications of metamodels will be worthwhile; preliminary efforts on this, for Alloy [6] and UML, can be found in [5]. We also intend to tie this work in with a refinement calculus that we have been creating for BON [4]. In this latter work, we have provided a formal semantics for much of BON in terms of predicates. Thus, we aim to define relationships between the BON metamodel – which captures syntactic constraints – and the formal semantics of abstractions and relationships described elsewhere.

References

1. B. Meyer. *Object-Oriented Software Construction*, Prentice-Hall, 1997.
2. *OMG Unified Modelling Language Specification 1.3*, OMG, June 1999.
3. S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. The PVS Language Reference Version 2.3, SRI International Technical Report, September 1999.
4. R. Paige and J. Ostroff. An Object-Oriented Refinement Calculus. Technical Report CS-1999-07, York University, December 1999.
5. R. Paige and J. Ostroff. Precise and Formal Metamodelling with the Business Object Notation and PVS. Technical Report CS-2000-03, York University, August 2000.
6. M. Vaziri and D. Jackson. Some Shortcomings of OCL, the Object Constraint Language of UML. Technical Report, MIT Laboratory for Computer Science, December 1999.
7. K. Walden and J.-M. Nerson. *Seamless Object-Oriented Software Development,* Prentice-Hall, 1995.

The Metamodelling Language Calculus: Foundation Semantics for UML

Tony Clark[1], Andy Evans[2], and Stuart Kent[3]

[1] King's College London
anclark@dcs.kcl.ac.uk
[2] University of York
andye@cs.york.ac.uk
[3] University of Kent at Canterbury
s.j.h.k@ukc.ac.uk

Abstract. The Metamodelling Language (MML) is a sub-set of the Unified Modeling Language (UML) that is proposed as the core language used to bootstrap the UML 2.0 definition initiative. Since it is meta-circular, MML requires an external formal semantics in order to ground it. This paper defines the MML Calculus which is used to formally define MML and therefore provides a semantic basis for UML 2.0.

1 Introduction

The Unified Modeling Language [19] is a standardized graphical notation for expressing the structure and behaviour of object-oriented software systems. It is essentially a family of extensible modelling notations. The current UML definition lacks a number of desirable features that are currently being addressed through a co-ordinated effort to define a new version (UML 2.0). These features include enhancing the modularity and extensibility of UML and addressing the notion of UML semantics.

This paper describes the semantics of the MML Calculus which used as the basis for developing the MML metamodelling language. MML is the basis of a modular semantics-rich method called MMF [7] [5] which is being proposed by the pUML group as a framework for the definition of UML 2.0. MML is a language mainly aimed at meta-modellers who are familiar with UML. This paper deals with foundational semantic issues that enable MML to be a generic metamodelling language suitable for defining UML 2.0. Features which are outside the scope of this paper include: patterns for metamodelling; details of inheritance mechanisms; details of class instantiation; details of package extension; details of invariant checking. These issues are dealt with in [5].

The rest of this paper is structured as follows Section 2 defines the MML Calculus syntax and semantics. Section 3 defines the MML language in terms of the MML Calculus. The MML language is textual, each new construct is introduced and translated to the MML Calculus. Section 4 concludes by reviewing MML and describing the future directions of this work.

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 17–31, 2001.

2 The MML Calculus

The MML is based on a small language called the MML calculus. It is an imperative object-oriented calculus that captures the essential operational features of MML. The calculus is based on the ς-calculus of Cardelli and Abadi [6]. To define the MML Calculus the ς-calculus is extended with some basic data types (integer, string, boolean, set and sequence) and operations over values of these data types. The rest of this section is structured as follows. Section 2.1 defines the syntax of the MML Calculus. Section 2.2 defines the semantics of the MML Calculus. Section 2.3 defines how to encode functions in the calculus, these are required to define parameterized operations over models. Section 2.4 defines some builtin operations that are either convenient or which cannot be defined in the calculus.

2.1 Syntax

The core syntax and value domain of the MML Calculus is defined below:

$e, a, b ::=$	expression
v, w	variable
n	integer
b	boolean
s	string
$[v_i(w_i) = e_i]^{i \in [0,n]}$	object expression
$e.v$	field reference
$a.v := b$	attribute update
$a.v := (w)b$	method update
$\mathbf{Set}\{e_i\}^{i \in [0,n]}$	set expression
$\mathbf{Seq}\{\}$	empty sequence
$\mathbf{Seq}\{a \mid b\}$	pair expression

$x, y ::=$	value
n	integer
b	boolean
s	string
$[v_i \mapsto \alpha_i]^{i \in [0,n]}$	object
$\mathbf{Set}\{x_i\}^{i \in [0,n]}$	set
$\mathbf{Seq}\{\}$	empty sequence
$\mathbf{Seq}\{x \mid y\}$	pair
(v, ρ, e)	field closure

The core syntax will be extended with extra features as the description of the MML Calculus proceeds. This document uses the following conventions to define syntax: terminals are given in bold; X^* represents 0 or more occurrences of X; $X^?$ represents an optional X; $X|Y$ represents X or Y parentheses can be used to group elements (bold parentheses are terminals); $X^{i \in [n,m]}$ is a sequence of elements where X contains free occurrences of i.

An object expression defines a collection of fields. Each field has a name, a self parameter and a body. Like the ς-calculus we conflate the notion of *attribute* and *method* into a single *field*. Because attribute update occurs sufficiently often we distinguish between method and attribute update.

Appendix A defines the substitution of expressions for free variables. Variables are bound in methods via the self argument. When a field is referenced in an object, the object is supplied as the value of the self argument.

2.2 Semantics

For convenience we conflate the syntactic and value domains for integers, booleans, strings and empty sequences. We also make use of different categories of *environment* which are partial functions. An environment ϵ is extended with

an association between a key k and a value v to produce $\epsilon[k \mapsto v]$. Environments satisfy the following law:

$$(\epsilon[k_1 \mapsto v_1])[k_2 \mapsto v_2] = (\epsilon[k_2 \mapsto v_2])[k_1 \mapsto v_1] \text{ when } k_1 \not= k_2$$

An object is an environment that maps field names to addresses. A lexical environment maps variables to values. A heap h is an environment that maps addresses to field closures. A field closure (v, ρ, e) contains a self variable v, a body e and a lexical environment ρ mapping the free variables of e to values.

The semantics of the MML Calculus us given by a relation $h, \rho, e \Rightarrow x, h'$ which tells us the result x and final heap h' produced by performing expression e with respect to a starting heap h in the context of a lexical environment ρ. The relation is defined as follows:

$$h, \rho[v \mapsto x], v \Rightarrow x, h \tag{1}$$

$$h, \rho, k \Rightarrow k, h \tag{2}$$

$$h, \rho, [v_i(w_i) = e_i]^{i \in [1,n]} \Rightarrow [v_i = \alpha_i]^{i \in [1,n]}, h[\alpha_i \mapsto (w_i, \rho, e_i)]^{i \in [1,n]} \text{ fresh } \alpha_i \tag{3}$$

$$\frac{\begin{array}{c} h, \rho, e \Rightarrow x, h'[\alpha \mapsto (w, \rho', b)] \\ h'[\alpha \mapsto (w, \rho', b)], \rho'[w \mapsto x], b \Rightarrow y, h'' \end{array}}{h, \rho, e.v \Rightarrow y, h''} \; x = [v_i = \alpha_i, v = \alpha]^{i \in [1,n]} \tag{4}$$

$$\frac{h, \rho, e \Rightarrow x, h'}{h, \rho, e.v := (w)b \Rightarrow x, h'[\alpha \mapsto (w, \rho, b)]} \; x = [v_i = \alpha_i, v = \alpha]^{i \in [1,n]} \tag{5}$$

$$\frac{\begin{array}{c} h, \rho, e \Rightarrow x[v \mapsto f], h' \\ h', \rho, b \Rightarrow y, h'' \end{array}}{h, \rho, a.v := b \Rightarrow y, h''[\alpha \mapsto (w, [v \mapsto y], v)]} \tag{6}$$

$$\frac{h_i, \rho, e_i \Rightarrow x_i, h_{i+1} \;\; i \in [1, n]}{h_1, \rho, \text{Set}\{e_i\}^{i \in [1,n]} \Rightarrow \text{Set}\{x_i\}^{i \in [1,n]}, h_{n+1}} \tag{7}$$

$$\frac{\begin{array}{c} h, \rho, a \Rightarrow x, h_1 \\ h_1, \rho, b \Rightarrow y, h_2 \end{array}}{h, \rho, \text{Seq}\{a|b\} \Rightarrow \text{Seq}\{x|y\}, h_2} \tag{8}$$

Free variables are bound to values in the current lexical environment (1). Constant expressions k are integers, booleans, strings and the empty sequence. When evaluated, a constant expression produces itself as a value (2). An object expression denotes an object. Fresh heap addresses are used for the object's fields. Notice that the current lexical environment is captured by each field since it contains the bindings for all free variables in a field body (3). Field reference causes the field body to be evaluated with respect to its lexical environment (4). Method update (5) replaces a field with a delayed expression. Field update (6) replaces a field with the value of an expression. The component expressions of a set expression are all evaluated and then the set is created. Sets do not live in the heap and therefore set operations do not cause side-effects (7). A pair is created after evaluating its head and tail (8).

MML is implemented as a Java program called MMT (the metamodelling tool). MMT runs a virtual machine that executes the MML Calculus. The machine is defined by transforming the relation $h, \rho, e \Rightarrow, x, h'$ into a transition function over machine states such that $([], \rho, [e], h, ()) \mapsto^* ([x], \rho, [], h', ())$. A machine state has the form (s, ρ, c, h, d) where the new components are: a stack s for intermediate values; a control c that is used as an instruction stream; a dump d that is used to save and restore machine contexts during field reference. A prototype version of MMT is available at [27].

2.3 Functions

The kernel calculus is object-based and not function-based like the λ-calculus. However, functions can be easily embedded in the calculus thereby providing the best of both worlds. A function with an argument v and a body b is $\lambda v.e$:

$$\begin{array}{lll} e, a, b ::= \ldots \text{as before} & & \text{expression} \\ & \lambda v.e & \text{function expression} \\ & \textbf{let } v = a \textbf{ in } b \textbf{ end} & \text{local definition} \end{array}$$

A function is an object with structure: $[\text{arg(self)} = \text{self}; \text{val(self)} = b[\text{self.arg}/v]]$ An application expression has the form $a(b)$ where a is an expression denoting the operator and b is an expression denoting the operand. The following equivalence is used: $a(b) = (a.copy.arg := b).val$ The copying (see section 2.4) is required in case the function is recursive.

The **let** expression introduces local definitions. Each definition consists of a variable v and a value a. The **let** expression has a body b. The scope of the variable is the body of the **let**. A **let** expression is defined in terms of a function: $\textbf{let } v = a \textbf{ in } b \textbf{ end} = (\lambda v.b)(a)$

2.4 Builtin Operations

The definition of MML relies on a number of builtin methods and operators. Extra rules (like δ-rules for the λ-calculus) are added to the calculus in order to define these features. Most of the rules involve functions, such as + or 'and' that do not depend on or change the current lexical environment or heap. The syntax is defined as follows:

$$\begin{array}{lll} e, a, b ::= \ldots \text{as before} & & \text{expression} \\ & \oplus {e_i}^{i \in [1,n]} & \text{operator expression} \end{array}$$

The semantics for each operator $\oplus$ is given by a rule $\text{Op}(\oplus)$:

$$\frac{h_i, \rho, e_i \Rightarrow x_i, h_{i+1} \; i \in [1,n]}{h_1, \rho, \oplus {(e_i)}^{i \in [1,n]} \Rightarrow \oplus {(x_i)}^{i \in [1,n]}, h_{n+1}} \; \text{Op}(\oplus_n) \tag{9}$$

The following binary operations are builtin: Op(+); Op(−); Op(∗); Op(/); Op(and); Op(or); Op(xor); Op(implies) When an object is copied, fresh addresses (α'_i) are allocated for its fields. Note that the copy is *shallow*, i.e. the values associated with the fields are not copied:

$$\frac{h, \rho, e \Rightarrow [v_i \mapsto \alpha_i]^{i\in[1,n]}, h[\alpha_i \mapsto (w_i, \rho_i, e_i)]^{i\in[1,n]}}{h, \rho, e.\text{copy} \Rightarrow [v_i \mapsto \alpha'_i]^{i\in[1,n]}, h[\alpha_i \mapsto (w_i, \rho + i, e_i)]^{i\in[1,n]}}$$

Copying for atomic values and sets has no effect. This is expressed by a generic rule Copy(x) and a collection of rules for all integers n Copy(n), for all booleans b Copy(b), for all strings s Copy(s) and for all collections c Copy(c):

$$\frac{h, \rho, e \Rightarrow x, h'}{h, \rho, e.\text{copy} \Rightarrow x, h'} \text{ Copy}(x)$$

Operations on sets are based on two operations: non-deterministic selection and adjoining an element to a set. Set extension is an operation that is constructed from set adjoin. Basic set operations are Op(select) and Op(adjoin):
$\text{select}(\text{Set}\{x_i\}^{i\in[1,n]}) = x_j$ for some $j \in [1, n]$
$\text{adjoin}(x, \text{Set}\{x_i\}^{i\in[0,n]}) = \text{Set}\{x, x_i\}^{i\in[0,n]}$
The head and tail of sequences are accessed using the following operations Op(head) and Op(tail): $\text{head}(\text{Seq}\{h|t\}) = h$ and $\text{tail}(\text{Seq}\{h|t\}) = t$. Equality Op(=) is defined as a builtin operation that returns true when atomic values are the same, objects have the same fields, when sets have the same elements and when sequences are either both empty or have equal heads and tails. Otherwise equality returns false. More sophisticated notions of equality can be constructed as methods for classes of MML object.

Boolean values support conditional commands by defining a function that evaluates a consequent c or alternative a: true = [if(_) = $\lambda c.\lambda a.c.val$] and false = [if(_) = $\lambda c.\lambda a.a.val$] The calculus is extended with a conditional expression:

$e, a, b ::=$	... as before	expression
	if e **then** a **else** b **end**	conditional expression

The conditional expression is defined as follows: **if** e **then** a **else** b **end** = e.if([val(_) = a])([val(_) = b])

3 The Metamodelling Language

The metamodelling language is given a semantics by a translation to the MML Calculus. MML is meta-circular, in the sense that it represents all the types and operations in order to describe its own operation. This section defines how the following MML features are represented in the MML Calculus: objects (section 3.1); methods (section 3.2); types (section 3.3); classes (section 3.4); method invocation (section 3.5); OCL (section 3.6); packages (section 3.7). The meta-circular MML model is shown in figure 1.

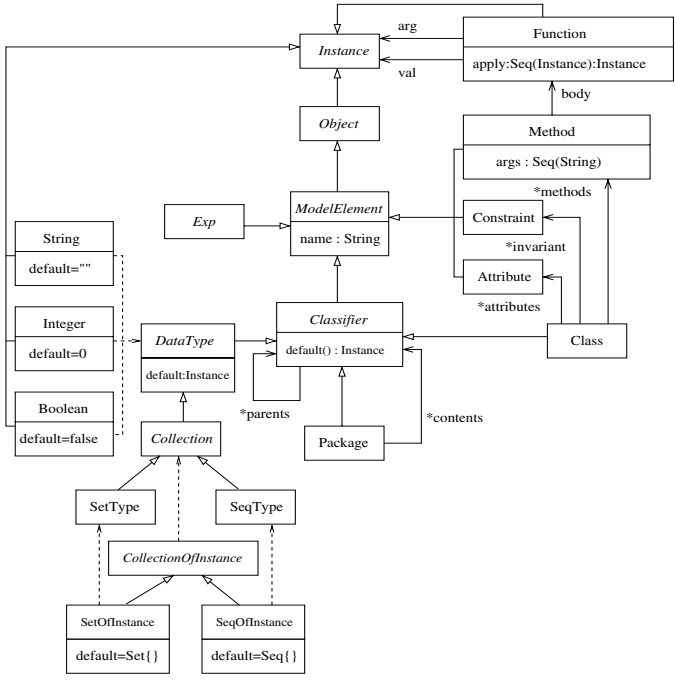


Fig. 1. The MML Language Model

3.1 Objects and Atomic Expressions

MML supports integer; boolean; string; set and sequence expressions. All of the builtin operators defined in section 2.4 are supported by MML. The semantics of the following MML expressions is given by a direct translation to the MML Calculus:

$m, l ::=$	MML expression
v, w	variable
n	integer
b	boolean
s	string
$\oplus {m_i}^{i \in [1,n]}$	operator expression
$m.v := l$	field update
$\mathbf{Set}\{m_i\}^{i \in [0,n]}$	set expression
$\mathbf{Seq}\{\}$	empty sequence expression
$\mathbf{Seq}\{l \mid m\}$	pair expression
$\mathbf{let}\ v = a\ \mathbf{in}\ b\ \mathbf{end}$	local definition
$\mathbf{if}\ e\ \mathbf{then}\ a\ \mathbf{else}\ b\ \mathbf{endif}$	conditional expression

All data values in MML have a field named 'of' whose value is a *classifier* for the data value. A classifier contains type information for a collection of its instances.

MML knows directly about a number of builtin classifiers. Access to the builtin classifiers is defined by the generic rule Of:

$$\frac{h, \rho, m \Rightarrow x, h'}{h, \rho, m.\text{of} \Rightarrow y, h'} \ \text{Of}(x, y)$$

which is used to construct rules for every integer n $\text{Of}(n, \text{Integer})$, for every boolean b $\text{Of}(b, \text{Boolean})$, for every string s $\text{Of}(s, \text{String})$, for every set s $\text{Of}(s,$ SetOfInstance) and for every sequence s $\text{Of}(s, \text{SeqOfInstance})$. The classifiers Boolean, String, SetOfInstance and SeqOfInstance are MML objects that define the appropriate data types.

MML object expressions require that the object's classifier be supplied. Each MML field is a MML calculus field and may optionally supply the self parameter w_i. If the self parameter is not supplied then it defaults to 'self':

$$\begin{array}{rll} l, m ::= & \ldots \text{ as before} & \text{MML expression} \\ & @m\ v_i(w_i) = m_i\ \textbf{end}^{i \in [0,n]} & \text{object expression} \end{array}$$

where m denotes the object's classifier. The expression is translated to an MML calculus expression by inserting the 'of' field with a dummy self parameter:

$$[\text{of} = (\_)m; v_i(\text{self}) = m_i]^{i \in [0,n]}$$

3.2 Methods

MML methods are parameterized OCL expressions whose values are computed when a message is sent to an object. The methods are based on the representation of functions given in section 2.3; the representation is extended so that functions are classified as instances of the class Function and to allow functions with multiple arguments. Methods extend functions with an extra implicitly defined argument for 'self'. A function applies arguments using the following method:

```
Function::apply(args : Seq(Instance)):Instance
    if args = Seq{} then self.val
    else (self.copy.arg := args.head).val.apply(args.tail)
    endif
```

The following syntax is used to denote MML functions:

$$\begin{array}{rll} m, l ::= & \ldots \text{ as before} & \text{MML expression} \\ & \textbf{fun}\ (v^*)\ m\ \textbf{end} & \text{function} \\ & \textbf{meth}\ (v^*)\ m\ \textbf{end} & \text{method} \end{array}$$

Unary functions are translated to MML objects, multary functions are curried and methods insert an extra argument named 'self':

$$\begin{array}{l} \textbf{fun}\ (v)\ m\ \textbf{end} = @\text{Function arg(self)} = \text{self}; \text{val}(f) = m[f.\text{arg}/v]\ \textbf{end} \\ \textbf{fun}\ (v_1, \ldots, v_n)\ m\ \textbf{end} = \textbf{fun}\ (v_1)\ \ldots\ \textbf{fun}\ (v_n)\ m\ \textbf{end}\ \ldots\ \textbf{end} \\ \textbf{meth}\ (v_1, \ldots, v_n)\ m\ \textbf{end} = \textbf{fun}\ (\text{self}, v_1, \ldots, v_n)\ m\ \textbf{end} \end{array}$$

Methods are defined by classes and invoked by sending an MML object a message. Message delivery involves looking the method up via the object's classifier and then invoking the method with respect to the object and the arguments. This is explained in section 3.5:

$$\begin{aligned} l, m ::= & \ldots \text{as before MML expression} \\ & m.v(l_i^{i \in [0,n]}) \text{ send expression} \end{aligned}$$

3.3 Types and Type Expressions

All MML data values have classifiers; given a value x, the classifier of x is x.of. A classifier may be user defined (such as Animal or Factory) or may be defined as part of the MML language (such as String and Boolean). MML makes a distinction between *data types* that classify non-object values and *classes* that classify object values. This section describes the basic infrastructure of data types and their denotation.

All classifiers define a collection of methods and invariants for their instances. Each classifier must specify a default value to be used when a new slot using the classifier as a type is created; this is initially specified as Instance and is redefined in concrete sub-classes of Classifier. There are two main sub-classes of Classifier: Class and DataType. Instances of DataType classify non-object data values. DataType redefines the default method to return the value of the default attribute.

3.4 Class Expressions

MML classes are defined using *class expressions*. A class expression is sugar for object expressions and simply serves to capture the common pattern of class definition. Since MML is meta-circular, all classes are objects, all meta-classes are objects and so on. Class definition syntax is defined as follows:

$l, m ::=$	... as before $\mid c \mid k$	MML expressions
$c ::=$	**class** $v\mu^?\pi^?(\alpha\|\nu\|\sigma)^*\iota^?$ **end**	class definition
$\mu ::=$	**metaclass** m	metaclass
$\pi ::=$	**extends** $m(,m)^*$	parents
$\alpha ::=$	$v : \tau$	attribute
$\nu ::=$	$v(\delta^?(,\delta)^*)m$	method
$\sigma ::=$	$v = m$	slot
$\delta ::=$	$v : \tau$	declaration
$\tau ::=$		type
	v	type name
	Set(τ)	set type
	Seq(τ)	seq type
$\iota ::=$	**inv** $(sm)^*$	invariant

A class expression consists of the name of the class followed by a number of clauses. A class is an object with its own classifier. The classifier is expressed in a class expression in the optional *metaclass* clause; if it is omitted then it

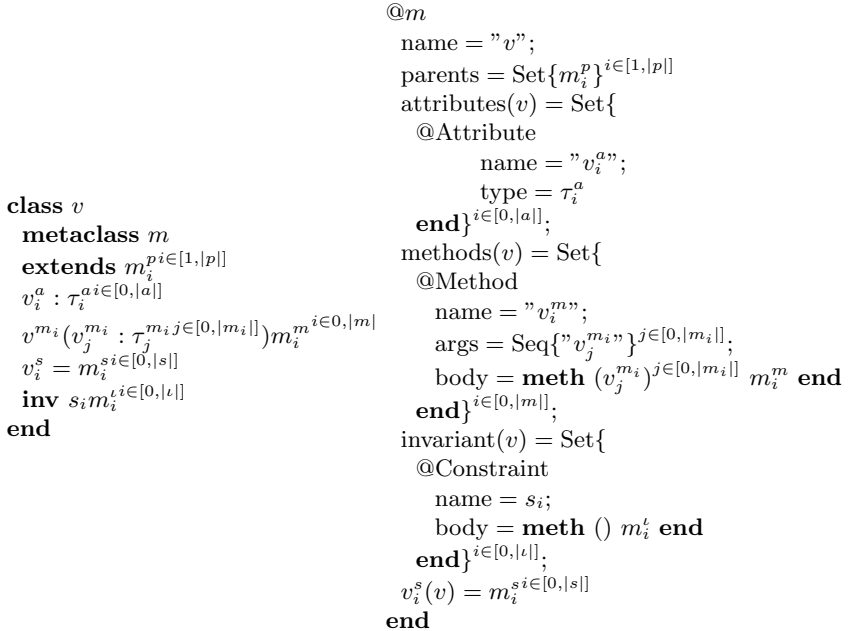


Fig. 2. Translation of Class Definition to Object Expression

defaults to Class. A class has multiple parents from which it inherits various definitions. The parents of a class are expressed in the optional *parents* clause; if it is omitted then the class will have the single parent Object. A class contains a number of definitions for attributes, methods and slots in the *definitions* clause. The attributes define the slots contained in instances of the class and the methods define the behaviour of the instances. The slot definitions allow extra information to be added to the class being defined where there is no syntax support. Extra slots are required when using a non-standard meta-class.

Figure 2 defines the translation of an MML class definition to an MML object expression. We will consider each feature category in turn; where appropriate we will draw attention to the scope of names available when each category is evaluated.

There are $|p|$ parent expressions. The scope of the new class name v does not include the parents since it is illegal to create cycles in the inheritance structure. There are $|a|$ attributes. The scope of v includes the attribute definitions since a class can contain an attribute whose values are instances of the class. There are $|m|$ methods, Each method has m_i arguments. The body of the method is a method function. The scope of v contains the method definitions since the methods of a class can refer to that class. Note that the arguments of the method functions are defined in an inner scope so they may shadow the class name

v. There are $|\iota|$ constraints defined for the invariant of a class. The scope of v includes the invariant. There are $|s|$ slots. The scope of v includes the slot values. The slots should correspond to attributes of the meta-class m that are not explicitly introduced by the class definition transformation.

3.5 Method Invocation

Method invocation in MML occurs when a send expression is performed. The expression has the following syntax:

$$\begin{aligned} l, m ::= &\ldots \text{as before} \quad \text{MML expression} \\ &m.v({l_i}^{i \in [0,n]}) \text{ send expression} \end{aligned} \tag{10}$$

where m is the target, v is the method name and l_i are arguments. A method with the name v is found by searching through the methods defined by the classifier of m.

Message delivery occurs by invoking the message delivery service. This is implemented directly in the calculus, but it is convenient to think of it as a method defined by the classifier of the target[1]. The method is defined as follows:

```
Classifier::send(target:Instance,message:String,args:Seq(Instance)):Instance
  let methods = self.allMethods()→select(m | m.name = message)
  in if methods = Seq{} then methods.head.apply(Seq{target | args})
     else self.error("no method for " + message)
     endif end
```

3.6 The Object Constraint Language

The Object Constraint Language (OCL) is an expression language used to express invariants, pre- and post- conditions, and guards on state transitions. OCL is fully integrated within MML by using the builtin operations defined by the Op rule 9 and by reducing collection operations to a very small number of primitives and then using these to implement methods in MML classifiers. OCL thereby becomes a convenient syntax for invoking a collection of predefined operations and methods whose implementation can be understood in terms of a very small number of primitive operations.

In this section we extend MML with the rest of OCL expression syntax. Figure 1 shows the root of the OCL expression hierarchy Exp. Each new syntactic category involves a new method definition for the appropriate classifier and a translation from the OCL expression to the appropriate method call. There are three groups of definitions: generic collections; sets; sequences. Bags are currently not implemented in MML but should follow the same implementation pattern as sets and sequences.

[1] If this approach was implemented efficiently then it would provide a mechanism for controlling message delivery at the meta-level

Collections. Many OCL operations work on all types of collection. Typically these operations are implemented in terms of lower-level operations that are specific to the particular type of collection:

$$\begin{array}{lll} l, m ::= \ldots \text{as before} & & \text{MML expression} \\ \quad m \rightarrow v(m(,m)^*)^? & & \text{collection operation} \\ \quad m \rightarrow v(w|l) & & \text{quantified operation} \\ \quad m \rightarrow \text{iterate}(vw = m|m) & & \text{iteration} \end{array}$$

Collection operation names are: size, includes (an element), count (ocurrences of an element), includesAll (elements of another collection), isEmpty, notEmpty, sum (the elements). Quantified operations are exists and forAll. The class CollectionOfInstance is the root of the collection type hierarchy. Its implementation is an example of how MML supports multiple meta-levels. Its meta-class Collection is a sub-class of DataType and therefore is the class of all collection types.

There are two different categories of expression: those that introduce new variables (exists, forAll and iterate) and those that do not. Expressions that do not introduce new variables are translated to send expressions in MML. For example $m \rightarrow \text{includes}(l)$ becomes $m.\text{includes}(l)$, with the following implementation:

CollectionOfInstance::includes(o : Instance) : Boolean
  self→ count(o) > 0
CollectionOfInstance::count(o : Instance) : Integer
  self→ iterate ($v_1 v_2 = 0|$ **if** v_1.equals(o) **then** $v_2 + 1$ **else** v_2 **endif**)

Expressions that introduce variables are translated to send expressions that bind the variables using one or more functions. Functions are objects in MML and may be passed as arguments to methods. The following table shows the translations:

$$\begin{array}{rl} m \rightarrow \text{exists}(v|l) & m.\text{exists}(\textbf{fun}(v)\ l\ \textbf{end}) \\ m \rightarrow \text{forAll}(v|l) & m.\text{forAll}(\textbf{fun}(v)\ l\ \textbf{end}) \\ m_1 \rightarrow \text{iterate}(v_1 v_2 = m_2|m_3) & m_1.\text{iterate}(\textbf{fun}(v_1, v_2)\ m_3\ \textbf{end}, m_2) \end{array}$$

The quantified operations are defined in terms of iterate. Iterate is abstract at the collection level and is implemented by each concrete type of collection.

CollectionOfInstance::exists(f : Function) : Boolean
  self→iterate(e a = false | a or f.apply(Seq{e}))
CollectionOfInstance::forAll(f : Function) : Boolean
  self→iterate(e a = true | a and f.apply(Seq{e}))

There is nothing special about the collection method definitions. MML is open to extension just like any other object-oriented model. The combination of polymorphism, dynamic binding and first class functions provides a very powerful abstraction and extension mechanism.

Sets. Set expression syntax is the same as that for collections. Set operations are: union; intersection; −; including (adjoin); symmetricDifference; asSequence. Parametric operations are: select (positive filter); reject (negative filter); collect (map). In each case the parametric operations are translated to send expressions with a function argument.

The class SetOfInstance is a sub-class of CollectionOfInstance and implements set operations. Many of the methods are implemented in terms of other SetOfInstance methods. The only set operations that rely on primitive operations are including that uses adjoin and iterate that uses select. These primitives along with the empty set can be viewed as being the essence of sets in MML. The following shows the definition of iterate:

```
SetOfInstance::iterate(v:Instance,f:Function):Instance
 if self = Set{} then v
 else let e = self.select
      in (self→excluding(e)).iterate(f.apply(Seq{e,v}),f) end
 endif
```

Sequences. OCL sequence expressions are implemented in MML and the MML Calculus using the same pattern of features as defined in the previous section. OCL sequences ultimately depend upon two primitive operations: head and tail. These are builtin to the OCL Calculus as defined in section 2.4.

3.7 Package Expressions

A package in MML is a container of classes and packages similar to those defined by the Catalysis approach [12]. Each class and package has a name and navigation expressions can be used to extract package contents. A package is itself a classifier; it is used to classify collections of objects. It is beyond the scope of this paper to deal with this feature of MML in detail; as a classifier, a package must have parents, methods and invariants. A package is created using a *package expression*:

$$\begin{array}{lll} l, m ::= \text{... as before} \mid p & & \text{MML expressions} \\ p ::= \quad \textbf{package}\ v\mu^{?}\pi^{?}(c|k|p)^{*}(\nu|\sigma)^{*}\iota^{?}\ \textbf{end} & & \text{package definition} \end{array}$$

A package definition expression is translated to an object expression in the MML Calculus. This translation is the same as that shown in figure 2 except that a package has an extra field called contents whose value is a set of classifiers. Each classifier defined by the package is also a field of the package; this allows us to navigate to package elements using names. The name of the package is scoped over each of the contents; this allows the contents to be mutually recursive.

MML consists of a collection of packages. It is beyond the scope of this paper to describe the package structure of MML. Consider a package called concepts that defines the core modelling concepts in MML. Figure 3 shows how such a package is expressed in MML and given a semantics by a translation to the MML Calculus; the details of class translation are omitted, but follow the definition given in figure 2.

```
package concepts
  class Classifier
  ...
  end
  class Package
    extends Classifier
  ...
  end
  class Class
    extends Classifier
  ...
  end
end
```

```
[of(concepts) = concepts.Package;
 name(_) = "concepts";
 contents(concepts) = Set{
   concepts.Classifier,
   concepts.Package,
   concepts.Class};
 Classifier(concepts =
   [of(_) = concepts.Class;
    name(_) = "Classifier"; ...];
 Package(concepts =
   [of(_) = concepts.Class;
    name(_) = "Package";
    parents(_) = Set{concepts.Classifier}; ...];
 Class(concepts =
   [of(_) = concepts.Class;
    name(_) = "Class";
    parents(_) = Set{concepts.Classifier}; ...];
```

Fig. 3. Package Translation

4 Conclusion

This paper has defined the MML Calculus which is used as the basis for a meta-circular modelling language called ML. The aim of MML is to provide a sound basis for UML 2.0. The need for a precise semantics for UML and OCL is shown by the increasing number of papers describing sophisticated tools and techniques. The following is a small selection: [1], [16], [23]. It is essential that the UML 2.0 initiative provide a standardized formal semantics for a UML core. The work described in this paper is part of an ongoing initiative by the pUML Group (`http://www.puml.org`) [7], [5], [8], [9], [27], [13]. The approach separates syntax and semantics so that the syntax of UML may change (for example graphical OCL [15], [18]) while the semantic domain remains the same.

We have taken a translational approach to the semantics of MML. This provides a semantics in terms of a much smaller calculus thereby achieving a parsimony of concepts. Such an approach has a distinguished history leading back to Landin and Iswim. We would hope that this approach will lead to proof systems and refinement calculi for MML. The approach should be contrasted with other metamodelling methods and tools including: BOOM [20] and BON [21]. The key novel feature of MML is that is aims to be exclusively based on UML and OCL; this allows UML to be both an extension and an instance of MML. UML is based on the Meta-Object Facility (MOF) which is a relatively small meta-model. MOF is not based on a precise semantics; it is hoped that work on MML will influence the future development of MOF.

It is somewhat difficult to validate the semantics since all definitions of UML are very imprecise; most parts of UML 1.3 are defined as syntax only. We can

report that the implementation of MML based on the calculus in MMT has been used to develop MML programs of about 3000 lines of code; based on this empirical evidence we can tentatively claim that MML behaves as expected.

The semantics described in this paper is imperative and operational. In that sense it is richer than UML whose informal semantics is *declarative*. However, the MML Calculus has a strictly declarative sub-language which captures the same semantics as other model-based approaches to UML semantics such as [24] and those that use Z.

The use of meta-circular language definitions has a distinguished history, for example CLOS [2], ObjVLisp [3] [4] [11] and Smalltalk [14] use meta-classes. The theory of meta-classes is discussed in [17]. The use of meta-models to describe UML and OCL is the accepted technique [22].

This is ongoing work. Core MML is nearing completion and a tool, MMT, that supports definition, use and checking of MML is under development. We intend to consolidate MML in the near future, for example by adding types [10], [25]; then, MML will be used to define a number of languages that contribute to the UML family.

References

1. Bottoni P., Koch M., Parisi-Presicce F., Taentzer G. (2000) Consistency Checking and Visualization of OCL Constraints. In Evans A., Kent S., Selic B. (eds) UML 2000, LNCS 1939, 278 – 293, Springer-Verlag.
2. Bobrow D. (1989) Common Lisp Object System Specification. Lisp and Symbolic Computation, 1(3/4).
3. Briot J-P, Cointe P. (1986) The ObjVLisp Model: Definition of a Uniform Reflexive and Extensible Object-oriented Language. In Proceedings of ECAI 1986.
4. Briot J-P., Cointe P. (1987) A Uniform Model for Object-oriented Languages Using the Class Abstraction. In proceedings of IJCAI 1987.
5. Brodsky S., Clark A., Cook S., Evans A., Kent S. (2000) A feasibility Study in Rearchitecting UML as a Family of Languages Using a Precise OO Metamodelling Approach. Available at `http://www.puml.org/mmt.zip`.
6. Cardelli L., Abadi M. (1996) A Theory of Objects. Springer-Verlag.
7. Clark A., Evans A., France R., Kent S., Rumpe B. (1999) Response to IML 2.0 Request for Information. Available at `http://www.puml.org/papers/RFIReponse.PDF`.
8. Clark A., Evans A., Kent S. (2000) The Specification of a Reference Implementation for UML. Special Issue of L'Objet.
9. Clark A., Evans A., Kent S. (2000) Profiles for Language Definition. Presented at the ECOOP pUML Workshop, Nice.
10. Clark A. (1999) Type-checking OCL Constraints. In France R. & Rumpe B. (eds) UML '99 LNCS 1723, Springer-Verlag.
11. Cointe, P. (1987) Metaclasses are First Class: the ObjVLisp Model. In Proceedings of OOPSLA 1987.
12. D'Souza D., Wills A. C. (1998) Object Components and Frameworks with UML – The Catalysis Approach. Addison-Wesley.

13. Evans A., Kent S. (1999) Core metamodelling semantics of UML – The pUML approach. In France R. & Rumpe B. (eds) UML '99 LNCS 1723, 140 – 155, Springer-Verlag.
14. Goldberg A., Robson D. (1983) Smalltalk-80: The Language and Its Implementation. Addison Wesley.
15. Howse J., Molina F., Kent S., Taylor J. (1999) Reasoning with Spider Diagrams. IEEE Symposium on Visual Languages '99, 138 – 145. IEEE CS Press.
16. Hussmann H., Demuth B., Finger F. (2000) Modular Architecture for a Toolset Supporting OCL In Evans A., Kent S., Selic B. (eds) UML 2000, LNCS 1939 LNCS, 278 – 293 , Springer-Verlag.
17. Jagannathan S. (1994) Metalevel Building Blocks for Modular Systems. ACM TOPLAS 16(3).
18. Kent S. (1997) Constraint Diagrams: Visualizing Invariants in Object-Oriented Models. In Proceedings of OOPSLA '97, 327 – 341.
19. Object Management Group (1999) OMG Unified Modeling Language Specification, version 1.3. Available at `http://www.omg.org/uml`.
20. Overgaard G. (2000) Formal Specification of Object-Oriented Metamodelling. FASE 2000, LNCS 1783.
21. Paige & Ostroff (2001) Metamodelling and Conformance Checking with PVS. To be presented at FASE 2001.
22. Richters M., Gogolla M. (1999) A metamodel for OCL. In France R. & Rumpe B. (eds) UML '99 LNCS 1723, 156 – 171, Springer-Verlag.
23. Richters M., Gogolla M. (2000) Validating UML Models and OCL Constraints. In Evans A., Kent S., Selic B. (eds) UML 2000 LNCS 1939, 265 – 277, Springer-Verlag.
24. Richters M., Gogolla M. (2000) A Semantics for OCL pre and post conditions. Presented at the OCL Workshop, UML 2000.
25. Schurr A. (2000) New Type Checking Rules for OCL (Collection) Expressions. Presented at the OCL Workshop at UML 2000. Available from `http://www.scm.brad.ac.uk/research/OCL2000`.
26. Warmer J., Kleppe A. (1999) The Object Constraint Language: Precise Modeling with UML. Addison-Wesley.
27. The MMT Tool. Available at `http://www.puml.org/mmt.zip`.

A Substitution for Free Variables

$$v[e/w] = \begin{cases} e \text{ when } v = w \\ v \text{ otherwise} \end{cases} \quad \text{variable}$$

$$k[e/v] = k \quad \text{atomic expression}$$

$$[v_i(w_i) = e_i]^{i\in[1,n]}[e/v] = [(v_i(w_i) = e_i)[e/v]]^{i\in[1,n]} \quad \text{object expressions}$$

$$(v(w) = a)[b/w'] = \begin{cases} v(w) = a & \text{when } w' = w \\ v(w) = (a[b/w']) & \text{otherwise} \end{cases} \quad \text{method}$$

$$a.v[b/w] = (a[b/w]).v \quad \text{field reference}$$

$$(a.v := b)[e/w] = (a[e/w]).v := (b[e/w]) \quad \text{field update}$$

$$(a.v := (w)b)[e/w'] = \begin{cases} (a[e/w']).v := (w)b & \text{when } w' = w \\ (a[e/w']).v := (w)(b[e/w']) & \text{otherwise} \end{cases} \quad \text{method update}$$

$$\text{Set}\{e_i\}^{i\in[1,n]}[e/v] = \text{Set}\{e_i[e/v]\}^{i\in[1,n]} \quad \text{set expression}$$

$$\text{Seq}\{a|b\}[e/v] = \text{Set}\{a[e/v]|b[e/v]\} \quad \text{pair}$$

Compositional Checking of Communication among Observers*

Ralf Pinger and Hans-Dieter Ehrich

Abteilung Informationssysteme, Technische Universität Braunschweig
Postfach 3329, D-38023 Braunschweig, Germany
{R.Pinger|HD.Ehrich}@tu-bs.de
Fax: +49-531-3913298

Abstract. Observers are objects inside or outside a concurrent object system carrying checking conditions about objects in the system (possibly including itself). In a companion paper [EP00], we show how to split and localise checking conditions over the objects involved so that the local conditions can be checked separately, for instance using model checking. As a byproduct of this translation, the necessary communication requirements are generated, taking the form of RPC-like action calls (like in a CORBA environment) among newly introduced communication symbols. In this paper, we give an algorithmic method that matches these communication requirements with the communication pattern created during system specification and development. As a result, correctness of the latter can be proved. In case of failure, the algorithm gives warnings helping to correct the communication specification.

Keywords. compositionality, distributed logic, model checking, modelling and design, object system, temporal logic, verification.

1 Introduction

In this paper, we elaborate on a novel compositional checking technique for distributed object systems that may be used for deductive verification, model checking or testing. In a companion paper [EP00], we show how to split and localise checking conditions over the objects involved so that the local conditions can be checked separately. Here we show how the necessary communication and thus the overall object system can be checked.

An object system in our sense is a community of sequential objects operating concurrently and communicating via synchronous RPC-like message passing (like in a CORBA environment). Objects may be distributed over sites.

The complexity of system verification is exponential in the number of concurrent components: the system state space "explodes" in size. To overcome this problem, compositional specification and verification techniques have been widely discussed. The idea is best expressed in [dR97]:

* This work was partially supported by the EU under ESPRIT IV Working Group 22704 ASPIRE.

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 32–44, 2001.

> "The purpose of a compositional verification approach is to shift the burden of verification from the global level to the local, component, level, so that global properties are established by composing together independently (specified and) verified component properties".

The benefit of compositional verification is obvious: instead an exponential increase of the overall verification amount, compositional verification has a linear complexity with respect to the number of components.

The first compositional verification approach was given by [MC81] where a rule for composing networks was given that is analogous to pre- and postconditions in sequential program proving. In [Pnu85], Pnueli defined the *assume-guarantee* approach that is very powerful but requires some human interaction for dividing the global property into suitable modular properties. Abadi and Lamport [AL93,AL95] used a modular way of specifying systems and were able to include liveness properties in the guarantee parts of the assume-guarantee rules. Moreover, their approach deals with fairness and hiding.

Clarke, Long and McMillan adapted these techniques for model checking [CLM89]. Their approach employs interfaces which represent only that part of a subsystem which is observable by one particular other subsystem. By composing such an interface with the corresponding subsystem, formulae can be verified in the smaller system which are true in the entire system. The performance of this approach depends heavily on the size of the interfaces, so it is best suited for loosely coupled systems. Another limitation deals with the fact that the interface rule can only handle boolean combinations of temporal properties of the individual processes.

Grumberg and Long as well as Josko developed machine supported versions of compositional verification. Grumberg and Long used a subclass of CTL which does not allow existentially quantified path formulae [GL94]. Another limitation is that all components of the system have to synchronise on a transition. This is suitable for clocked systems like, e.g., in hardware but it is not appropriate for software components which are distributed over a computer network or even over the internet. Josko [Jos89] used a modular but quite restricted version of CTL for writing assertions in an assume-guarantee way. Moreover, the number of verification steps grows exponentially with the number of until/unless formulae in the assumptions.

Fiadeiro and Maibaum give foundational aspects of object-oriented system verification in [FM95]. They have developed sound inference rules for reasoning about object system. However, there is no tool support for this approach.

The idea we put forward here does not restrict the logic: we employ a distributed extension of full CTL for specifying the checking conditions. The added "distributed" facility consists of nesting statements about communication with concurrent objects. This logic, called D_1, can be automatically translated to the logic D_0 that also employs full CTL locally but has a very restricted communication mechanism reflecting RPC-like action calling.

Our method is orthogonal to the other compositionality approaches, it may be combined with the others. Which combinations are useful is subject to further

study. In a sense, traditional compositionality approaches can be embedded into our approach: the traditional global view may be seen as bound to an "external observer" in our sense.

Object-oriented modelling and design is widely accepted; this is given evidence, among others, by the fact that UML has become an industrial standard. The work reported here is performed in the context of developing the TROLL language, method and tools for information systems development [HDK+97,DH97, GKK+98] where we put some effort in investigating the logic and mathematical foundations of the approach. The semantics of TROLL is based on distributed temporal logic like the one used in this paper. Its use for specification is described in [ECSD98,EC00].

In the next section, we give a brief and informal introduction to our approach, details can be found in [EP00]. We use the example given there in order to demonstrate how to distribute "global" assertions over the objects involved, introducing communication requirements in the form of RPC-like communication rules. Here we capitalise on a result on translating distributed logics published in [ECSD98,EC00]. In the 3rd section we present the main result of this paper, an algorithmic method that matches these communication requirements with the communication pattern created during system specification and development.

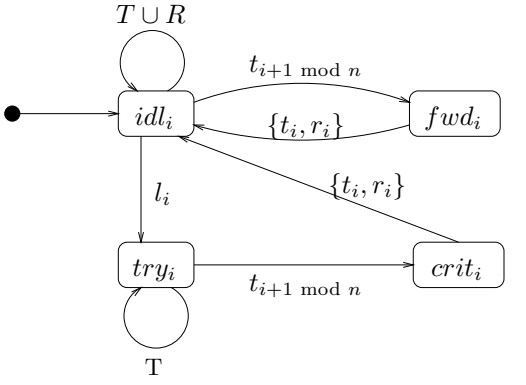
2 Distributing Checking Conditions

The idea is illustrated by an n-process mutual exclusion example where we identify an object with a process. A process may be idle or trying to enter the critical region; only one process at a time is allowed to enter its critical region.

Example 1 (Mutual Exclusion). We present a "self-organising" version of the n-process mutual exclusion problem that is advantageous in cases where the system is at rest most of the time because then there is no communication traffic. In the beginning, all processes $P_0, \ldots, P_{n-1}$ are idle, and the system is at rest. When k processes try to enter the critical region, $0 < k \leq n$, one is chosen nondeterministically and a token ring of all processes is initialised, giving all processes in turn a chance to enter the critical region if they want to. Whenever all processes are idle, the system may go at rest again, i.e., the token ring is put dormant.

A state diagram of P_i is shown in figure 1. Each process $P_i, i \in \{0, \ldots, n-1\}$ has four states: $\{idl_i, try_i, crit_i, fwd_i\}$. Process P_i has the token iff it is in one of its right-hand side states, $crit_i$ or fwd_i, respectively. A transition carrying a set of actions as a label is an abbreviation for a set of transitions between the same pair of states, one for each action in the set.

The token ring is established as follows. Each process P_i receives the token from its right-hand neighbour $P_{i+1 \bmod n}$ and sends it to its left-hand neighbour $P_{i-1 \bmod n}$. Transmitting the token is modelled by two actions for each process P_i: $t_{i+1 \bmod n}$ represents receiving the token, and t_i represents forwarding it. Of course, forwarding in P_i synchronises with receiving in $P_{i-1 \bmod n}$, represented



$$T = \{t_j \mid j \not= i+1 \bmod n\}, R = \{r_k \mid k \not= i\}$$

Fig. 1. State diagram of process i

by sharing the same action t_i. The communication between these processes is easily expressed by action calling formulae: $P_k.(t_i \rightarrow P_j.t_i)$ with $k \not= j$.

On receiving the token, a process enters the critical region if it wants to, i.e., if it is in its *try* state. Otherwise, i.e., if it is in its *idl* state, it simply forwards the token, moving to the *fwd* state when receiving the token and back to the *idl* state when forwarding it. The token is also forwarded when the process leaves the critical region, going into its *idl* state. The process may move from its *idl* state to its *try* state via action l_i any time without synchronising with other processes, indicating its spontaneous desire to enter the critical region.

When the system is at rest, one or more processes may spontaneously enter their *try* states, and one of these will proceed into the critical region, making sure by direct communication with all the others that they keep away. At the same time, the token ring is put into operation. Whenever a process terminates its critical region or forwarding state and all others are idle, the system may go at rest again, i.e., all processes are idle and the token ring is dormant. This is the case when the last active process P_i chooses its reset transitions r_i, synchronising with the r_i reset transitions of all other processes. The latter are only applicable in their *idl* states, ensuring that all processes are in their *idl* states after the move.

For the sake of simplicity, each process synchronises with all the others when entering the critical region, also when the token ring is active when this is not really necessary. In our solution shown in figure 1, this is accomplished by the loops at the *idl* and *try* states, synchronising with token actions of all the other processes. Also, we abstain from enforcing that the system goes at rest again as soon as it can, the token ring may nondeterministically remain active for a while (or even forever).

We concentrate on checking safety.

$$P_i.\,\mathsf{A}\,\mathsf{G}(crit_i \;\Rightarrow \bigwedge_{j=1, j \not= i}^{n} P_j.\neg\, crit_j) \;\text{ for all }\; i \in \{0, \ldots, n-1\}$$

The assertions are written in the distributed version of CTL that we introduce below. It should be intuitively clear what they mean. The i-th safety condition makes an assertion about all the other processes: they must not be in the critical region when the i-th process is.

In order to check safety, and assuming that the communication is correct, all we have to do is to prove the following.

$$P_i.(crit_i \;\Rightarrow\; t_{i+1 \bmod n}) \quad \text{for all } i \in \{0,\dots,n-1\}, \text{ and}$$
$$P_j.(t_{i+1 \bmod n} \;\Rightarrow\; \neg\, crit_j) \text{ for all } j \not= i,\ j \in \{0,\dots,n-1\}$$

The first condition says that P_i cannot be in the critical region unless it has been entered via action $t_{i+1 \bmod n}$. The second condition says that if $t_{i+1 \bmod n}$ has happened, no other process can be in the critical region. These conditions are local, they are obviously satisfied by our solution.

We note in passing that also other correctness criteria (cf. [HR00]) are satisfied by our solution: *liveness* (each process which is in its try state will eventually enter the critical section), *non-blocking* (a process can always request to enter the critical region) and *no-strict-sequencing* (processes need not enter the critical region in strict sequence). We do not elaborate on these conditions here. A compositional proof for the liveness property can be found in [EP00].

It remains to prove that communication indeed works correctly. What must be shown is the following.

$$P_i.(a_i \;\Rightarrow\; P_j.a_i) \text{ and } P_j.(a_i \;\Rightarrow\; P_i.a_i) \text{ for } \; i,j \in \{0,\dots,n-1\}.$$

where a may stand for t or r: each token or reset action synchronises with its counterparts in the other processes. For $i = j$, the formulae are trivial. The precise meaning of these formulae will be made clear in the next section.

Observers are objects inside or outside a concurrent object system carrying checking conditions about objects in the system (possibly including itself). Let I be a finite set of observers (or, rather, observer identifiers). Each observer $i \in I$ has an individual set A_i of action symbols expressing which action happens during a given transition, and an individual set Σ_i of state predicate symbols expressing which attributes have which values in a given state. Let $\Sigma = (I, \{A_i, \Sigma_i\}_{i\in I})$ be the observer signature. Our multi-observer logic is like the distributed logic D_1 described in [ECSD98,EC00] but instantiated with CTL, the computation tree logic due to E. Clarke and A.E. Emerson [CE81]. This logic $\mathsf{D}_1(\Sigma)$ — or D_1 for short when Σ is clear from context — is defined as follows.

Definition 1. The *observer logic* D_1 is the I-indexed family of local logics $\{\mathsf{D}_1^i\}_{i\in I}$ where, for each $i \in I$, D_1^i is CTL over A_i and Σ_i with the added possibility of using formulae in $\mathsf{D}_1^j, j \in I, j \not= i$ as subformulae.

A given D_1 formula $i.(\dots j.\psi \dots)$ with a communication subformula $j.\psi \in \mathsf{D}_1^j, j \not= i$, means that i communicates — or synchronises — with j and asserts that ψ is true for j at this moment of synchronisation. Thus, each D_1 formula

is bound to an observer, and each formula of another observer may serve as a subformula, representing communication with that observer.

For instance, the safety formulae in example 1 are in D_1: the subformulae $P_j.\neg\, crit_j, j \in \{1,\ldots,n\}, j \not= i$, are communication subformulae.

As shown in example 1 above, these assertions are transformed to purely local formulae and primitive communication formulae,

$$P_i.(a_i \;\Rightarrow\; P_j.a_i) \text{ and } P_j.(a_i \;\Rightarrow\; P_i.a_i) \text{ for } \; i,j \in \{0,\ldots,n-1\}.$$

All these formulae are in D_1. The latter communication subformulae characterise RPC, the basic middleware communication mechanism.

Formulae of this kind constitute the sublogic D_0 which we define below.

First we give a more detailed and more precise definition of D_1 semantics. We assume a family $\mathcal{M} = \{\mathcal{M}_i\}_{i\in I}$ of models to be given, one for each observer, where $\mathcal{M}_i = (S_i, \rightarrow_i, L_i), i \in I$. For each observer i, S_i is its set of states, $\rightarrow_i \subseteq S_i \times A_i \times S_i$ is its action-labelled state transition relation, and $L_i : S_i \rightarrow_i 2^{\Sigma_i}$ is its state labelling function. We write $s \stackrel{a}{\rightarrow}_i t$ for $(s,a,t) \in \rightarrow_i$. As in example 1, we may extend the model to allow for finite sets B of actions as transition labels: $s \stackrel{B}{\rightarrow} s' \;\Leftrightarrow\; \forall a \in B : s \stackrel{a}{\rightarrow} s'$.

We assume that the reader is familiar with the conventional CTL semantics (cf., e.g., [HR00]): it defines the meaning of $\mathcal{M}_i, s_i \models_i \varphi$ for every observer i where $s_i \in S_i$ and $\varphi \in \mathsf{D}_1^i$, as long as φ does not contain any communication subformula — and as long as there are no action symbols, i.e. $|A_i| = 1$ for every observer i.

Action symbols are easily introduced by "pushing them into the states": given a model $\mathcal{M} = (S, \rightarrow, L)$ where $\rightarrow \subseteq S \times A \times S$ and $L : S \rightarrow 2^{\Sigma}$, we define the model $\mathcal{M}' = (S', \rightarrow', L')$, $\rightarrow' \subseteq S' \times S', L' : S' \rightarrow A_i \times 2^{\Sigma_i}$, by

$$\begin{aligned} S' &= S \times A \times S \\ (s_1 \stackrel{a_1}{\rightarrow} s_2) \rightarrow' (s_3 \stackrel{a_2}{\rightarrow} s_4) &\text{ iff } \; s_2 = s_3 \\ L'(s_1 \stackrel{a}{\rightarrow} s_2) &= (a, L(s_2)) \end{aligned}$$

for all $s_1, s_2, s_3, s_4 \in S'$ and all $a, a_1, a_2 \in A$. Of course, $s \rightarrow' t$ stands for $(s,t) \in \rightarrow'$. We use action symbols a also as action occurrence predicates meaning that action a occurs during a transition (in $\mathcal{M}$) or during a state (in $\mathcal{M}'$), respectively.

Our construction is different from the ACTL construction given in [dNV90] where a new state s_a is introduced on each transition labelled a. Our construction gives us states with action symbols as state predicates; this is essential for the translation of D_1 to D_0 as described in the following section.

Capturing communication is not quite so easy. Note that with treating the actions as described above, we now synchronise states rather than transitions. However, whether a state of one observer synchronises with a given state of another one does not only depend on the current state of the former but on what happened before in the latter. We refer to [EC00,EP00] for more details on this matter.

Now we demonstrate how D_1 formulae can be translated to D_0 formulae in a sound and complete way where $\mathsf{D}_0 \subseteq \mathsf{D}_1$ is a sublogic featuring sketched above. Full details of the translation as well as soundness and completeness proofs are given in [EC00]. We first introduce D_0, outline the translation rules, and show how the translation can be utilised to transform global assertions (in D_1) into multiple local assertions (in D_0). This way, the communication primitives and the local assertions for the objects involved are separated so that they together guarantee the original global assertion.

For defining D_0, we assume again that a family of object signatures $\Sigma = (I, \{A_i, \Sigma_i\}_{i \in I})$ is given which we do not show explicitly in our notation.

Definition 2. The *object sublogic* $\mathsf{D}_0 \subseteq \mathsf{D}_1$ is the I-indexed family of local logics $\{\mathsf{D}_0^i\}_{i \in I}$ where, for each $i \in I$, each formula in D_0^i is either a CTL formula over A_i, Σ_i and $@j, j \in I$, or a communication formula of the form $a_i \Rightarrow j.a_j, j \in I, j \not= i, a_i \in A_i, a_j \in A_j$.

The predicates $i.@j$ mean the same as $i.j.\neg\bot$ in D_1, namely that i communicates with j. But this must be given an extra symbol in D_0 because $i.j.\neg\bot$ is not a formula in D_0. Note that $i.@i$ is trivially true: each object synchronises with itself all the time.

Examples of D_0 formulae have been given above. For the straightforward proof that D_0 is indeed a sublogic of D_1, we refer to [EC00].

The idea of the translation is as follows. Let an assertion $\varphi \in \mathsf{D}_1^i$ be given saying, somewhere in context, that an assertion ψ holds for j,

$$\varphi \;\Leftrightarrow\; i.(\ldots j.\psi \ldots),$$

where $j.\psi$ is in D_0, i.e., ψ does not have a communication subformula. The meaning is that i transmits a "message" q to j,

$$\overline{\varphi} \;\Leftrightarrow i.(\ldots q \ldots),$$

that is equivalent in j to the validity of ψ during communication with i,

$$\delta \;\Leftrightarrow j.(q \Leftrightarrow @i \wedge \psi)$$

q is a new action symbol in i as well as in j, i.e., there are local actions $i.q$ in i and $j.q$ in j "calling" each other. This communication is captured by

$$\alpha \;\Leftrightarrow i.(q \Rightarrow j.q)$$
$$\beta \;\Leftrightarrow j.(q \Rightarrow i.q)$$

The resulting formulae δ, α and β are in D_0. $\overline{\varphi}$ may still be in $\mathsf{D}_1 - \mathsf{D}_0$, but the number of communication subformulae has decreased by one. With α and β as defined above, the translation step is sound (φ implies $\overline{\varphi}$ and δ) and complete ($\overline{\varphi}$ and δ imply φ).

The translation of arbitrary D_1 formulae is accomplished by iterating the above step inside–out until no communication subformulae are left. Together

with the corresponding RPC-like communication formulae like α and β, the communication infrastructure is established for reducing global to local model checking. Clearly, the translation process terminates. Its soundness and completeness is proven in [EC00].

We illustrate the idea by the mutual exclusion example (example 1).

Example 2 (Mutual Exclusion cont.). In order to keep the example small but not too trivial, we take $n = 3$. The table in figure 2 shows how the safety condition for P_0 is translated.

Iteration ...
0 states the safety formula for P_0;
1 replaces the 1st communication subformula by q_{01} and defines q_{01} for P_1;
2 replaces the 2nd communication subformula by q_{02} and defines q_{02} for P_2.

The translation can be iterated for P_1 and P_2 respectively.

iteration	object	local formulae	communication
0	P_0	$\mathsf{A}\,\mathsf{G}(crit_0 \Rightarrow P_1.\neg crit_1 \wedge P_2.\neg crit_2)$	
1	P_0	$\mathsf{A}\,\mathsf{G}(crit_0 \Rightarrow q_{01} \wedge P_2.\neg crit_2)$	$q_{01} \Rightarrow P_1.q_{01}$
	P_1	$q_{01} \Leftrightarrow @P_0 \wedge \neg crit_1$	$q_{01} \Rightarrow P_0.q_{01}$
2	P_0	$\mathsf{A}\,\mathsf{G}(crit_0 \Rightarrow q_{01} \wedge q_{02})$	$q_{02} \Rightarrow P_2.q_{02}$
	P_2	$q_{02} \Leftrightarrow @P_0 \wedge \neg crit_2$	$q_{02} \Rightarrow P_0.q_{02}$

Fig. 2. P_0 safety condition transformed to D_0

Note that new action symbols are introduced along with the translation. However, "new" in this context means that the symbols have not been in the formula before but may very well have been in the object signature. In practice where we have implemented the system and want to check assertions in retrospect, this means that we have to look for suitable communication symbols that are already implemented. In this sense, the translation provides the "communication requests" that must be matched by the communication as modelled in system design. We elaborate on this idea in the next section.

3 Verifying Checking Conditions

The method for translating D_1 to D_0 formulae sketched above, generates new communication symbols as shown in figure 2.

In general, we have to find a mapping of the generated communication formulae α, β to the modeled action calling terms of the model. This operation can be automatised when combined with model checking in the following way. Recall the above D_1 formula $\varphi \Leftrightarrow i.(\ldots j.\psi \ldots)$ and the local D_0 formula $\delta \Leftrightarrow j.(q \Leftrightarrow @i \wedge \psi)$ which was generated during the translation. Note that we use the transformed transition model, in which the labels of the states include action symbols as state predicates. Now we compute by model checking the set S' of states of object j in which ψ holds. Let $B_j^{S'}$ be the set of action symbols that are in the labels of each state in S'. In order to compute the possible set of actions that imply the

formula ψ in j, we have to remove from $B_j^{S'}$ the set of action symbols that occur in any label of states not in S'. We call this set B_j^{Ac}, the set of actions that are potentially able to communicate with i preserving the formula ψ.

The above formula δ is true for all action symbols of B_j^{Ac} establishing a communication between object i and object j. Let $B_j^{Ca} \subseteq B_j^{Ac}$ be the set of communication actions of object j that perform a communication with object i and satisfy ψ. Note that the set B_j^{Ca} is easily computed by intersecting the set of actions of the modeled action callings of object j with B_j^{Ac}: every action of B_j^{Ac} that occurs in the action calling terms of object j, is a communication action and thus belongs to the set B_j^{Ca}. The set B_j^{Ca} denotes the set of actions that perform a communication with i preserving ψ.

After checking this we have to check the generated D_0 formula $\overline{\varphi} \Leftrightarrow i.(\dots q \dots)$ for all $q \in B_i^{Ca}$. The set B_i^{Ca} includes all communication actions q of object i that call an action $\mathrm{b} \in B_j^{Ca}$ in j. Instead of checking $\overline{\varphi}$ for all actions q in B_i^{Ca} we replace q by the disjunction of all actions in B_i^{Ca}. By finding at least one communication action satisfying $\overline{\varphi}$, a ψ preserving communication between i and j can be established. Note that checking of $\overline{\varphi}$ and δ can be done on the local models of object i and j; no global state transition graph is needed to verify these formulae.

```
Check-D0(φ̄, ψ: D0 formula; i, j: I): bool ;
  begin
    S' := Compute_States (j, ψ);
    B_j^S' := ⋃ {a | a = L_|Ai(s), s ∈ S'};
    B_j^Ac := B_j^S' − {a | a = L_|Ai(s), s ∈ S̄'};
    B_j^Ca := B_j^Ac ⋂ {a | j.(a ⇒ i.b)};
    B_i^Ca := {b | i.(b ⇒ j.a), a ∈ B_j^Ca};
    replace (φ̄, q → ⋁ B_i^Ca);
    return Mod_Check (i, φ̄);
  end ;
```

Fig. 3. Algorithm for verifying D_0 formulae

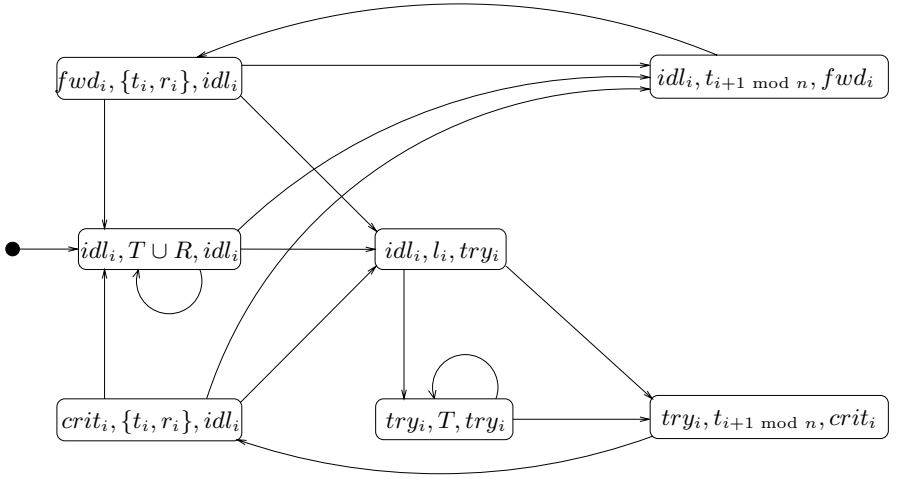
Figure 3 gives an algorithmic notation of the algorithm outlined above. The procedure Compute_States is a subroutine that is usually used in model checkers for computing the set of states satisfying a given formula. The replace procedure replaces the generated action symbol q with the disjunction of all action symbols of set B_i^{Ca}. If the set B_i^{Ca} is empty, then q would be false. The function $L_{|A_i}(s)$ is an abbreviation for $L(s) \cap A_i$. The procedure Mod_Check can be seen as a usual model checking procedure.

The overall truth value of the global D_1 formula depends on the truth values of the local D_0 formulae $\overline{\varphi}$ and δ and the correct mapping of the generated communication formulae α and β with the modeled communication requests, which is done by computing the sets B_i^{Ca} and B_j^{Ca}. It may happen that one of the above action sets is empty; that would be the case if there is no state in j satisfying ψ or if there is no suitable mapping from the computed actions

to modeled actions. In this case no communication between i and j can be established that satisfies ψ in j; the generated action formula q is replaced by false to indicate that there is no ψ preserving communication to j. If so, detailed warnings should be given to the user helping to find the error in the design.

By iterating the above algorithm for every D_1 communication subformula from the innermost communications to the outermost ones, the successful computation of S' and thus a valid procedure call for the model checker is guaranteed.

Example 3 (Mutual Exclusion cont.). To illustrate the above algorithm, we first have to translate the original model to the model with actions as state predicates. Figure 4 shows the model of process i after the translation. The value of the labeling function L_i is denoted by the second and the third item of each state label. The notation $\{t_i,\ r_i\}$ as a label in the states is an abbreviation for two states; usually every state has exactly one transition in the labelling. Correspondingly, the notation $T \cup R$ is also an abbreviation.



$$T = \{t_j \mid j \not= i + 1 \bmod n\}, R = \{r_k \mid k \not= i\}$$

Fig. 4. Translated model for the mutual exclusion example

Before giving an example for the algorithm, we refer to the D_0 specification generated in the previous section (figure 2). Iteration 1 produces the new action symbol q_{01}. The occurrence of symbol q_{01} should be equivalent to the formula $@P_0 \wedge \neg\, crit_1$ in process P_1. Following the algorithm we have to compute the set of states satisfying $\neg\, crit_1$ in P_1. This is true for all states except for $(try_1, t_2, crit_1)$. In the next step we compute the set of actions that are in the label of all states in S', obtaining $\{t_0, t_1, t_2, r_0, r_1, r_2, l_1\}$. By computing the set of potential communication actions, we have to reduce this set by all actions that are not in the labels of states of S'. In our example, t_2 is the only action that occurs outside of S'.

P_0	P_1	P_2
$P_0.(t_0 \rightarrow P_1.t_0)$	$P_1.(t_0 \rightarrow P_0.t_0)$	$P_2.(t_0 \rightarrow P_0.t_0)$
$P_0.(t_0 \rightarrow P_2.t_0)$	$P_1.(t_0 \rightarrow P_2.t_0)$	$P_2.(t_0 \rightarrow P_1.t_0)$
$P_0.(t_1 \rightarrow P_1.t_1)$	$P_1.(t_1 \rightarrow P_0.t_1)$	$P_2.(t_1 \rightarrow P_0.t_1)$
$P_0.(t_1 \rightarrow P_2.t_1)$	$P_1.(t_1 \rightarrow P_2.t_1)$	$P_2.(t_1 \rightarrow P_1.t_1)$
$P_0.(t_2 \rightarrow P_1.t_2)$	$P_1.(t_2 \rightarrow P_0.t_2)$	$P_2.(t_2 \rightarrow P_0.t_2)$
$P_0.(t_2 \rightarrow P_2.t_2)$	$P_1.(t_2 \rightarrow P_2.t_2)$	$P_2.(t_2 \rightarrow P_1.t_2)$

Fig. 5. Action calling terms of processes P_0, P_1, P_2

Now we continue with the modeled action callings to produce the set of communication formulae satisfying $\neg\, crit_1$ in j. The action callings of P_1 are given in figure 5. The only actions that perform communications with P_0 and match with the computed set of communication actions are $\{t_0, t_1\}$. Thus we have to find action callings in P_0 that communicate with P_1 via actions t_0 and t_1 as well. Due to the fact that P_0 has suitable communication requests that match with the computed actions, we have found a mapping between the generated communication requests of figure 2 and modeled action callings of figure 5. The overall truth value of $\overline{\varphi}$ can be computed by replacing q_{01} by the communication requests found by the algorithm. Repeating the above procedure for q_{02} we get the formula $\overline{\varphi} \;\Leftrightarrow\; \mathsf{A}\,\mathsf{G}\;\; (crit_0 \;\Rightarrow\; (t_0 \vee t_1) \wedge (t_1 \vee t_2))$. The truth of this formula is obvious in this case.

object	local formulae	communication
P_0	$\mathsf{A}\,\mathsf{G}(crit_0 \;\Rightarrow\; t_1)$	$t_1 \Rightarrow P_1.t_1$
		$t_1 \Rightarrow P_2.t_1$
	$t_2 \Leftrightarrow @P_1 \wedge \neg\, crit_0$	$t_2 \Rightarrow P_1.t_2$
	$t_0 \Leftrightarrow @P_2 \wedge \neg\, crit_0$	$t_0 \Rightarrow P_2.t_0$
P_1	$\mathsf{A}\,\mathsf{G}(crit_1 \;\Rightarrow\; t_2)$	$t_2 \Rightarrow P_2.t_2$
		$t_2 \Rightarrow P_0.t_2$
	$t_0 \Leftrightarrow @P_2 \wedge \neg\, crit_1$	$t_0 \Rightarrow P_2.t_0$
	$t_1 \Leftrightarrow @P_0 \wedge \neg\, crit_1$	$t_1 \Rightarrow P_0.t_1$
P_2	$\mathsf{A}\,\mathsf{G}(crit_2 \;\Rightarrow\; t_0)$	$t_0 \Rightarrow P_0.t_0$
		$t_0 \Rightarrow P_1.t_0$
	$t_1 \Leftrightarrow @P_0 \wedge \neg\, crit_2$	$t_1 \Rightarrow P_0.t_1$
	$t_2 \Leftrightarrow @P_1 \wedge \neg\, crit_2$	$t_2 \Rightarrow P_1.t_2$

Fig. 6. Local safety conditions and communication after matching

After iterating the above method for P_1 and P_2 we obtain a set of local safety conditions and action calling formulae as shown in figure 6. In the case of our example the algorithm computes a matching of the generated communication symbols $q_{i(i+1 \bmod 3)}$ and $q_{i(i-1 \bmod 3)}$ with the modeled actions $t_{i+1 \bmod 3}$ for $i \in \{0, 1, 2\}$ taken from the action calling terms.

With the matching of the generated communication symbols with the modeled action callings we are able to satisfy the local D_0 formulae (if possible). Due to the sound and complete translation of a given D_1 formula to D_0 formulae (cf. [EC00]), the truths of the global D_1 formula is implied by the truths of the local

D_0 formulae. Once we have found a matching satisfying the local D_0 formulae, the global D_1 formula is satisfied as well. Thus our reasoning from the local to the global level is sound due to the soundness and completeness of the D_1 to D_0 translation.

4 Concluding Remarks

The investigation of the idea presented in this paper is at its beginning, further study is necessary in order to explore how the idea can best be utilised in checking object system designs. For doing this, appropriate tool support in combination with existing model checkers is essential.

Although we introduce new communication conditions, the overall verification is promised to be very efficient. Due to the fact that the model checking complexity grows linearly with the length of a given CTL-formula (cf. [CGP00]), the local model checking amount is increased linearly by introducing new communication conditions. Compared to the verification of global models, the extra effort needed for checking newly introduced communication conditions on local models should be relatively small.

The technique described here has the limitation that we are able to deal with direct communication only. Note that there must be a direct communication between two components i and j, if e.g. i uses a communication subformula of observer j. Systems using indirect or asynchronous communication can not be verified directly: whereas indirect communication has to be split into its direct communication parts on the specification level, systems using asynchronous communication have to be transformed to synchronous ones. The transformation might be done by using special buffer objects.

The original motivation for studying D_1 and D_0 and their translation was to use them for modelling and specification [ECSD98]. D_0 was chosen to describe the semantics of the TROLL language, which was developed along with an application project to design an information system for a laboratory in the German National Institute of Weights and Measures (PTB) in Braunschweig, cf. [HDK+97,GKK+98].

References

[AL93] Martin Abadi and Leslie Lamport. Composing Specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[AL95] Martin Abadi and Leslie Lamport. Conjoining Specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. *Lecture Notes in Computer Science*, 131:52–71, 1981.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doran A. Peled. *Model Checking*. MIT Press, 2000.

[CLM89] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional Model Checking. In *Proceedings fo the 4th Annual Symposium on Principles of Programming Languages*, pages 343–362, 1989.

[DH97] G. Denker and P. Hartel. TROLL – An Object Oriented Formal Method for Distributed Information System Design: Syntax and Pragmatics. Informatik-Bericht 97–03, Technische Universität Braunschweig, 1997.

[dNV90] Rocco de Nicola and Frits Vaandrager. Action versus State based Logics for Transition Systems. In I. Guessarian, editor, *Semantics of Concurrency*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419, 1990.

[dR97] Willem-Paul de Roever. The Need for Compositional Proof Systems: A Survey. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 1–22, September 1997.

[EC00] H.-D. Ehrich and C. Caleiro. Specifying communication in distributed information systems. *Acta Informatica*, 36 (Fasc. 8):591–616, 2000.

[ECSD98] H.-D. Ehrich, C. Caleiro, A. Sernadas, and G. Denker. Logics for Specifying Concurrent Information Systems. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, pages 167–198. Kluwer Academic Publishers, 1998.

[EP00] H.-D. Ehrich and R. Pinger. Checking object systems via multiple observers. In *International ICSC Congress on Intelligent Systems & Applications (ISA'2000)*, volume 1, pages 242–248. University of Wollongong, Australia, International Computer Science Convetions (ICSC), Canada, 2000.

[FM95] Jose Luiz Fiadeiro and Tom Maibaum. Verifying for Reuse: Foundations of Object-Oriented System Verification. In C. Hankin, I. Makie, and R. Nagarajan, editors, *Theory and Formal Methods*, pages 235–257. World Scientific Publishing Company, 1995.

[GKK+98] A. Grau, J. Küster Filipe, M. Kowsari, S. Eckstein, R. Pinger, and H.-D. Ehrich. The TROLL Approach to Conceptual Modelling: Syntax, Semantics and Tools. In T.W. Ling, S. Ram, and M.L. Lee, editors, *Proc. of the 17th Int. Conference on Conceptual Modeling (ER'98), Singapore*, pages 277–290. Springer, LNCS 1507, November 1998.

[GL94] Orna Grumberg and David E. Long. Model Checking an Modular Verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[HDK+97] P. Hartel, G. Denker, M. Kowsari, M. Krone, and H.-D. Ehrich. Information systems modelling with TROLL formal methods at work. In *Information Systems*, volume 22, pages 79–99, 1997.

[HR00] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science - Modelling and reasoning about systems*. Cambridge University Press, 2000.

[Jos89] Bernhard Josko. Verifying the Correctness of AADL Modules using Model Checking. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 386–400, 1989.

[MC81] Jayadev Misra and K. Mani Chandy. Proofs of Networks of Processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, July 1981.

[Pnu85] Amir Pnueli. In Transition From Global to Modular Temporal Reasoning about Programs. In Krzysztof R. Apt, editor, *NATO ASI Series*, volume F13, 1985.

Combining Independent Specifications

Joy N. Reed[1*] and Jane E. Sinclair[2]

[1] Oxford Brookes University, Oxford, UK
Email: JNReed@brookes.ac.uk Fax: +44 (0) 1865 483666
[2] University of Warwick, Coventry, UK
Email: jane@dcs.warwick.ac.uk Fax: +44 (0) 2476 573024

Abstract. We present a formal framework for characterising plug-in relationships between component specifications. A suitability requirement is defined based on the effect one component has on the other in terms of deadlock. Unlike monotonic operations such as parallel composition, not all such suitability requirements are preserved by refinement. Hence, we define the notion of a bicompositional relation between co-operating processes which is preserved by component-wise refinements. The approach is described in CSP using the failures semantic model. The aim is to underpin a mixed-paradigm approach combining different specification methods, including state-based deductive formalisms such as Action Systems, and event-based model checking formalisms such as CSP/FDR. The objective is to play to the strengths and overcome limitations of each technique, by treating different system aspects with individual tools and notations which are most appropriate.

1 Introduction

A formal method is a mathematically-based theory which is used to describe and reason about the behaviour of a computer system. Application of a formal method encompasses specification of the system in a chosen formal notation, analysis and verification of key properties and stepwise refinement to a correct implementation. It is generally recognised that even partial use of these techniques during development can significantly increase the quality of software and hardware systems, with respect to correctness and maintainability. For example, the application of a general-purpose specification notation such as Z [Spi92] has been found to lead to the earlier discovery of design flaws. Formal modelling and verification of small security protocols such as that by Lowe and Roscoe [LR97], Lowe [Low96] and Meadows [Mea94] has revealed previously unsuspected flaws in the operation of these protocols.

Various formal methods with different theoretical bases have been proposed. No one formalism is fully suitable for all aspects of industrial-sized applications, as we have illustrated by directly comparing strengths and weaknesses of state-based, deductive reasoning approaches and event-based, model checking approaches applied to a distributed mail system [RSG99] and general routing

* This work was supported in part by the US Office of Naval Research

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 45–59, 2001.

protocols [RSR99]. With a deductive reasoning approach, a specification gives an abstract description of the significant behaviour of the required system. This behaviour can be verified for the defined implementation by proving the theorems which constitute the rules of refinement. With model checking, a specification corresponds to a formula or property which can be exhaustively evaluated on a specific finite domain representing implementations. Deductive reasoning is more general, but only partially automatable. Model checking is more limited but fully automatable. Our previous work [RSG99] shows that in addition to theoretical limitations of a notation, its form leads towards specification of a certain style and often with particular implicit assumptions.

Combining the different views can give a fuller picture, but the question remains as to how this integration can best be achieved. Incorporating all aspects into one unified approach based around a particular formalism and tool set is a possibility, but this could result in additional complexity and obscurity for loosely coupled components. In contrast, our approach for combining different formalisms is to relate their different system views in a way which allows meaningful and independent analysis, including stepwise development through separate refinement. Our long-term aim is to provide a formal underpinning for this approach, so that certain state-based safety properties can be handled with state-based techniques and tools, and event-based, liveness properties can be handled with model checkers. In this paper we identify relationships between loosely-coupled components which ensure the soundness of component-wise refinement. The notation used is CSP [Hoa85] which is particularly convenient for event-based specification and model checking. We indicate how this could be combined with a state-based notation such as Action Systems [BKS83], using their common failures divergences semantics.

2 Interoperating Components: An Example

In an environment of distributed systems it is increasingly the case that any transaction requires the interoperation of a chain of components and services which combine to produce (hopefully) the desired result. Various components may be selected as "off the shelf" products to plug-in to our particular application. Some of these services may be beyond our control, but we still have expectations of harmonious behaviour. The correct operation of an application depends not only on the integrity of its own functions, but also on the components with which it interacts and on the interaction itself. Whilst formal verification of the entire system is not practical, there may be certain crucial interactions which warrant the extra care provided by formalism. Some components, such as security services, may have been verified in their own right and come with assurance of their behaviour. We are interested characterising such assurances.

As an example, we consider a specification for a secure database which answers requests for information from its subscription customers. The database and many of its operations can be described in a natural way using a state-based

approach. The system must also take into account the need for appropriate security controls. A number of useful algorithms (and implementations of these) already exist for such things, so our top level specification states its basic requirements and relies on these being satisfied by a suitable plug-in component, which as a separate concern may range from providing only simple confidentiality through to providing additional authentication and integrity. The top level specification simply needs to know that a task will be performed (such as, a common session key being distributed to both database and client), with no need to place constraints on the values it requires. We wish to treat the functional properties of the subscription database and the security protocol as separate units which can be further developed and verified separately in different ways as appropriate.

Using Action Systems, which describe both system state and the interaction of events, we can specify a suitable top-level for the database. Conditions such as clearance to access data can be succinctly captured and verified in this way. In contrast, other tasks such as developing and verifying a suitable key exchange protocol between the parties is not best-suited to such a notation. In fact, suitable protocols have already been described and verified in other ways, notably using CSP. We wish to "hand over" from the Action Systems to CSP for these. We consider below how this hand over can be achieved and why something more than the usual parallel composition is desirable.

3 Combining Notations in Practice

The trace/failures/divergences semantics models of CSP described in the next section provide a unifying semantics for Action Systems and CSP. Although a common semantics provides a theoretical link, it does not immediately solve the problems of using the notations together. The specifier is concerned with the effects of combining different parts of a specification and the constraints placed on each part by others. Translating the specifications to common ground is a possibility but, in general, prohibitively cumbersome. This has led to on-going research for specific areas where manageable proof conditions can be generated. Schneider and Treharne [TS99,TS] have worked with B and CSP defining a CSP driver to control operations described in B. Butler [But99] defines a general approach to combining B and CSP which merges CSP into B through translation. Both approaches provide useful insights into how the two notations can be brought together, yet neither are sufficient for our purpose. Like Schneider and Treharne, we wish to maintain the separation of notations. Like Butler, we wish to allow each to have a more general use. Our aim is to provide a framework in which "suitable" components can be "plugged" together and refined separately. In this paper, we explore a notion of "suitable plug-in".

The parallel composition operator can be used to combine CSP processes and/or Action Systems. So if P is an Action Systems specification for some aspect or component of a system, and Q is a CSP specification for another aspect or component, then $P \parallel Q$ represents their parallel combination with behaviour

well-defined (although possibly not easily understood), and any safety (that is, trace) property of either P or Q with respect to their common actions/events is preserved by $P \parallel Q$. In contrast to safety properties, liveness properties are not preserved by the $\parallel$ operator, as illustrated by Example 2 below. If Q is a plug-in for P then at the very least we require that it should respond in a way suitable for P and not cause P to deadlock when they are run in parallel.

We formulate our suitability requirements using CSP semantics shared by both Action Systems and CSP. The issue of specialising to the different notations is returned to in later sections. We first give a brief introduction to CSP. An overview of CSP syntax and the failures model is given in the Appendix.

4 CSP and FDR

CSP [Hoa85] models a system as a *process* which interacts with its environment by means of atomic *events*. Communication is synchronous: an event takes place precisely when both process and environment agree on its occurrence. This rather than assignments to shared variables is the fundamental means of interaction between agents. CSP comprises a process-algebraic programming language. A related series of semantic models capture different aspects of observable behaviours of processes: traces, failures and divergences. The simplest semantic model is the *traces* model which characterises a process as an *alphabet* $\alpha(P)$ of events, together with the set of all finite traces, $traces(P)$, it can perform. The traces model is sufficient for reasoning about safety properties. In the failures model [BHR84] a process P is modelled as a set of *failures*. A *failure* is a pair (s, X) for s a finite trace of events of $\alpha(P)$, and X a subset of events of $\alpha(P)$; $(s, X) \in failures(P)$ means that P may engage in the sequence s and then refuse all of the events in X. The set X is called a *refusal*. The failures model allows reasoning about certain liveness properties. More complex models such as failures/divergences[BR85] and timed failures/divergences [RR99] have more structures allowing finer distinctions to support more powerful reasoning. Traces and failures are also defined for Action Systems [Mor90,WM90,But92] providing a semantic link to CSP. For the rest of this paper, we restrict our discussion to the *failures* model.

We say that a process P is a refinement of process S ($S \sqsubseteq P$) if any possible behaviour of P is also a possible behaviour of S:

$$failures(P) \subseteq failures(S)$$

which tells us that any trace of P is a trace of S, and P can refuse an event x after engaging in trace s, only if S can refuse x after engaging in s.

Intuitively, suppose S (for "specification") is a process for which all behaviours it permits are in some sense acceptable. If P refines S, then any behaviour of P is as acceptable as any behaviour of S. S can represent an idealised model of a system's behaviour, or an abstract property corresponding to a correctness constraint, such as deadlock or livelock freedom. A wide range of correctness conditions can be encoded as refinement checks between processes. Mechanical refinement checking is provided by Formal Systems' model checker, FDR [For].

5 The Relationship between Components

We view our top-level specification as structured as a set of interoperable components which act in parallel. In particular, we are interested in components, such as the key exchange protocol which may be called upon to perform some task as part of a larger system. Although parallel composition can be used to combine any two processes, we would not regard every process as a *suitable* fulfilment of the requirements of the larger system. It is this additional concept of *suitability* that we wish to capture.

Example 1. Suppose process P makes a request (event a) to receive two keys (events $k1$ and $k2$). P does not care in which order the keys are obtained.

$$P = (a \rightarrow k1 \rightarrow k2 \rightarrow P) \,\Box (a \rightarrow k2 \rightarrow k1 \rightarrow P)$$

Process Q is a component which is chosen to distribute keys, and this happens to be specified as responding first with $k1$ and then with $k2$.

$$Q = a \rightarrow k1 \rightarrow k2 \rightarrow Q$$

It is expected that establishing the keys as specified by Q will be achieved by some more detailed algorithm which, with internal details hidden, refines Q. We regard Q as a plug-in to P, since their joint behaviour expressed by $P \parallel Q$ conforms to one of the possibilities allowed by P. Clearly not every process which returns keys should be regarded as a plug-in to P. For example, consider R:

$$R = a \rightarrow k1 \rightarrow k1 \rightarrow R$$

This time R does not interact with Q in a desirable way since it does not have an acceptable pattern of response, and the result is deadlock at the third step.

In general, Q will be suitable to plug in to P if it is prepared to co-operate with the pattern set out by P. If P allows an (external) choice then Q may resolve it. If P's actions are nondeterministic, Q must be prepared to deal with each possibility. This can be characterised in terms of deadlock: Q must not cause P to deadlock when they are run in parallel.

Further problems are encountered when considering refinement. In general, we expect that whenever a specification is good enough for some purpose, then so is any refinement. However, Example 2 shows that if we are dealing with *plug-in relationships* between component processes which ensure that their parallel combination describes desirable properties of our system, it does not follow that component-wise refinements are suitable according to the same criteria. That is, for a relation ρ on processes, if $P \rho Q$, $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$, then by monotonicity we know that $P' \parallel Q' \sqsubseteq P \parallel Q$ – but is not necessarily true that $P' \rho Q'$.

Example 2. Suppose P and Q are both defined as follows:

$$P = (x \rightarrow P) \sqcap STOP$$
$$Q = (x \rightarrow Q) \sqcap STOP$$

We observe that P and Q satisfy the relationship:

$$P \sqsubseteq P \parallel Q$$

(this relationship might appear desirable since it ensures that Q cannot cause any deadlock not also allowed by P). P and Q satisfy this property, but not refinements:

$$P' = x \rightarrow P' \qquad\qquad Q' = STOP$$

Clearly $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$ and yet $P' \not\sqsubseteq P' \parallel Q'$.

Example 2 shows that potential candidates for describing suitable plug-in relationships between components may not be preserved by refinement. This would be disastrous from the point of view of building systems with independently developed components. In this paper we concentrate on patterns of interaction between two processes P and Q which, as in Example 1 above, behave like a simple remote procedure call from P to Q. We regard Q as a *plug-in* to P if Q responds in a way which does not increase the opportunities for deadlock. Furthermore, we regard Q to be a *suitable plug-in* to P if for any component-wise refinements Q' and P', Q' is a plugin to P'. In the rest of the paper, we investigate how to formalise these notions. We first define a general property of relations which is useful for capturing the notion that refinements of processes inherit their parents' relationship.

Definition 1 (Bicompositional Relations)**.** *Let ϕ and ρ each be a relation on X. We say that ϕ is bicompositional with ρ iff for $x \phi y$, $x \rho x'$, $y \rho y'$, then $x' \phi y'$*

Example 3. Let R be the relation $<$ and S the relation which holds between x and y iff $y = x + 1$. Then R is bicompositional with S. This example also shows that the property is not symmetric since S is not bicompositional with R. In addition, it is neither reflexive nor transitive.

In general, relations are not bicompositional with themselves, for example, the relation $<$ is not bicompositional with $<$. Equivalence relations are bicompositional with themselves, though not in general bicompositional with arbitrary other relations.

6 Bicompositional Refinements

We can capture the notion that a given relationship ϕ between cooperating specifications is inherited by refinements by requiring ϕ to be bicompositional with $\sqsubseteq$. We say that ϕ is bicompositional whenever

$$P \phi Q \ \wedge \ P \sqsubseteq P' \ \wedge \ Q \sqsubseteq Q' \ \Rightarrow \ P' \phi Q'$$

We can make the relationship given in Example 2 bicompositional by requiring not only that $P \sqsubseteq P \parallel Q$ but also that P is deterministic. However this does

not offer a general solution to this refinement paradox; it defeats the purpose of refinement since all refinements of P must then in fact be equal to P. We might instead insist that P and Q always operate together in a deadlock free fashion. We cannot ensure this by simply requiring that each of P and Q is deadlock free, as illustrated by $P = \bigsqcap_T x \to P$ and $Q = \bigsqcap_T x \to Q$. P and Q are each deadlock free since each is willing to do some event of T, but $P \parallel Q$ can deadlock whenever they do not agree on their chosen events. However, if we also require $P \parallel Q$ to be deadlock free as well, Example 4 below ensures that any refinements P' and Q' inherit their parents' good behaviour and so cannot deadlock when they themselves are run in parallel.

Example 4. Processes P and Q are mutually deadlock free iff each of P and Q is deadlock free, and their parallel composition is deadlock free. Mutual deadlock freedom is bicompositional.

Proof. A process is deadlock free iff it refines the process DF which is always willing to do something in its alphabet Σ:

$$DF = \bigsqcap_\Sigma a \to DF$$

Let $DF \sqsubseteq P$, $DF \sqsubseteq Q$, and $DF \sqsubseteq P \parallel Q$. Also let $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$. Then it follows by monotonicity that $DF \sqsubseteq P'$, $DF \sqsubseteq Q'$, and $P \parallel Q \sqsubseteq P' \parallel Q'$, and $DF \sqsubseteq P' \parallel Q'$. □

Mutual deadlock freedom, though bicompositional, is too strong a property to require of cooperating applications Q and P for which Q responds to a one-time invocation from P. For example, Q may be a set-up process which P calls once and only once. Thus, $P \parallel Q$ will properly deadlock on their joint alphabet after Q finishes its work, whilst P carries on with other events not requiring any participation from Q. Or Q behaves as a remote procedure call to P, which may acceptably never invoke any services provided by Q. Indeed, we may wish to allow P itself to deadlock. Let us imagine that we want P to trigger Q by handing over some parameters, which Q processes, subsequently returning results back to P. P is in control, and may invoke Q arbitrarily, including never; Q is always required to be ready, and is willing to be invoked forever. What is required is a bicompositional relation between P and Q which implies that their parallel combination deadlocks on the intersection of their joint alphabet J only where P chooses. Then, if we refine P and Q with P' and Q', the parallel combination of P' and Q' deadlocks on the intersection of their joint alphabet only where P' chooses.

We begin with some notation, then define a relationship which we regard as characterising the notion that one process is as live as another. We illustrate that this relationship is not in general preserved by refinement, and require that such a relation qualify as a plug-in only if it is preserved by refinement. We finally define some stronger bicompositional relations which qualify as plug-ins.

Notation. For process P and set J of events, $\alpha(P)$ represents the alphabet of P, and $traces(P) \upharpoonright J$ represents the set of traces of P stripped of all events not in J, and $failures(P) \upharpoonright J$ has both traces and refusals of P stripped of any events

not in J. We regard Q as a plug-in to P iff Q is allowed to refuse to do all of J after trace S (thus causing deadlock) only if P can as well:

Definition 2. Q **is as live as** P, for $\alpha P \cap \alpha Q = J$ means

$$(s, J) \in \mathit{failures}(P \parallel Q) \upharpoonright J \Rightarrow (s, J) \in \mathit{failures}(P) \upharpoonright J$$

This relation constrains Q to deadlock only when P might, and in particular, if P is deadlock free, then $P \parallel Q$ is deadlock free. But it is not bicompositional. For the processes defined below, Q is as live as P (which both behave chaotically), but for the refinements, Q' is not as live as P'.

Example 5. $P = (\Box_R r \to P) \sqcap \mathrm{STOP}$ $\quad Q = (\sqcap_R r \to Q) \sqcap \mathrm{STOP}$
$P' = \Box_R r \to P$ $\quad Q' = \mathrm{STOP}$

We characterise a plug-in relationship:

Definition 3. Q **plugs-in** P, means

1. *Q is as live as P, and*
2. *for all P', Q', if $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$, then Q' is as live as P'.*

We now identify bicompositional relations between P and Q which imply that Q plugs-in to P. The first definition characterises interactions between processes P and Q whereby whenever P is ready to output a value x on the channel T to Q, then Q is ready to receive it.

Definition 4. Q ***listens on*** T ***to*** P, for $T \subseteq \alpha P \cap \alpha Q$ means

$$x \in T \wedge s \frown \langle x \rangle \in \mathit{traces}(P) \upharpoonright J \wedge s \in \mathit{traces}(Q) \upharpoonright J$$
$$\Rightarrow (s, \{x\}) \notin \mathit{failures}(Q) \upharpoonright J$$

Theorem 1. *The relation* listens on T to *is bicompositional.*

Proof. Assume

(1) $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$

(2) $x \in T \wedge s \frown \langle x \rangle \in \mathit{traces}(P) \upharpoonright J \wedge s \in \mathit{traces}(Q) \upharpoonright J$
$\Rightarrow (s, \{x\}) \notin \mathit{failures}(Q) \upharpoonright J$

We must show

$$x \in T \wedge s \frown \langle x \rangle \in \mathit{traces}(P') \upharpoonright J \wedge s \in \mathit{traces}(Q') \upharpoonright J$$
$$\Rightarrow (s, \{x\}) \notin \mathit{failures}(Q') \upharpoonright J$$

Assume the hypothesis of the implication. By (1) $s \frown \langle x \rangle \in \mathit{traces}(P) \upharpoonright J$ and $s \in \mathit{traces}(Q) \upharpoonright J$. Thus by (2), $(s, \{x\}) \notin \mathit{failures}(Q) \upharpoonright J$, and again by (1) $(s, \{x\}) \notin \mathit{failures}(Q') \upharpoonright J$. □

We next define a bicompositional property which characterises a process Q outputting along channel R whenever P wants. We do not want to overly constrain Q, that is, Q should be allowed to deadlock on their joint alphabet whenever P is willing to do so. If we require that P either must be prepared to accept any answer communicated by Q, or possibly deadlock – but not both simultaneously – then the relation is bicompositional. The following definition characterises processes P and Q whereby whenever P is ready to receive input from Q, Q is ready to send it. It says that whenever P is ready to receive any value of R – there is an obligation on P to be ready to receive any other value as well, and there is an obligation on Q be ready to output something.

Definition 5. Q answers on R to P, for $R \subseteq \alpha P \cap \alpha Q = J$ means

$$x \in R \wedge y \in R \wedge s \frown \langle x \rangle \in \mathit{traces}(P) \upharpoonright J \wedge s \in \mathit{traces}(Q) \upharpoonright J$$
$$\Rightarrow (s, R) \notin \mathit{failures}(Q) \upharpoonright J \wedge (s, \{y\}) \notin \mathit{failures}(P) \upharpoonright J$$

Theorem 2. *The relation* answers on R to *is bicompositional.*

Proof. Assume

(1) $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$

(2) $x \in R \wedge y \in R \wedge s \frown \langle x \rangle \in \mathit{traces}(P) \upharpoonright J \wedge s \in \mathit{traces}(Q) \upharpoonright J$
$\Rightarrow (s, R) \notin \mathit{failures}(Q) \upharpoonright J \wedge (s, \{y\}) \notin \mathit{failures}(P) \upharpoonright J$

Let

$$x \in R \wedge y \in R \wedge s \frown \langle x \rangle \in \mathit{traces}(P') \upharpoonright J \wedge s \in \mathit{traces}(Q') \upharpoonright J$$

We must show

$$(s, R) \notin \mathit{failures}(Q') \upharpoonright J \wedge (s, \{y\}) \notin \mathit{failures}(P') \upharpoonright J$$

Assume $(s, R) \in \mathit{failures}(Q') \upharpoonright J$. Then by (1), $(s, R) \in \mathit{failures}(Q) \upharpoonright J$, and furthermore, $s \frown \langle x \rangle \in \mathit{traces}(P) \upharpoonright J$ and $s \in \mathit{traces}(Q) \upharpoonright J$. Thus by (2), $(s, R) \notin \mathit{failures}(Q) \upharpoonright J$, and this contradiction establishes that $(s, R) \notin \mathit{failures}(Q') \upharpoonright J$. Assume $(s, \{y\}) \in \mathit{failures}(P') \upharpoonright J$. Again by (1), $(s, \{y\}) \in \mathit{failures}(P) \upharpoonright J$, but this contradicts (2) thereby establishing the theorem. □

The next theorem establishes that for processes P and Q synchronising on the intersection of their alphabets, if Q listens to P, and Q answers P, then Q plugs-in to P.

Theorem 3. *If Q listens on T to P and Q answers on R for $T \cup R = \alpha(P) \cap \alpha(Q) = J$, then $(s, J) \in \mathit{failures}(P \parallel Q) \upharpoonright J$ only if $(s, J) \in \mathit{failures}(P) \upharpoonright J$.*

Proof. Assume $(s, J) \in \mathit{failures}(P \parallel Q) \upharpoonright J$. By *failures* model definition for the parallel operator (see Appendix), for some refusal sets X, Y, $X \cup Y = J$, $s \in \mathit{traces}(P) \upharpoonright J$ and $s \in \mathit{traces}(Q) \upharpoonright J$ and $(s, X) \in \mathit{failures}(P) \upharpoonright J$, and $(s, Y) \in \mathit{failures}(Q) \upharpoonright J$. Assume $(s, J) \notin \mathit{failures}(P) \upharpoonright J$. Then $X \not= J$, and since P cannot refuse all of J there must exist an $x \notin X$ such that $s^\frown\langle x\rangle \in \mathit{traces}(P) \upharpoonright J$. There are two cases : $x \in T$ or $x \in R$. *Case 1.* Assume $x \in T$. Since Q listens on T to P, $(s, \{x\}) \notin \mathit{failures}(Q) \upharpoonright J$. Since $(s, Y) \in \mathit{failures}(Q) \upharpoonright J$, then by *failures* axiom (M3) which says that any subset of a refusal set is itself a refusal set, it follows that $x \notin Y$. This contradicts that $X \cup Y = J$, and case is proved. *Case 2.* Assume $x \in R$. Since Q answers on R to P, $(s, R) \notin \mathit{failures}(Q) \upharpoonright J$. Furthermore, $(s, \{y\}) \notin \mathit{failures}(P) \upharpoonright J$ for any $y \in R$. By (M3), it follows that $X \cap R = \{\}$. Hence $R \subseteq Y$ and again by (M3), $(s, R) \in \mathit{failures}(Q) \upharpoonright J$. This contradiction proves the case and the theorem. □

Summary. We have identified a notion of one process Q being as live as another process P, whereby we mean that Q introduces no more possibilities for deadlock than P. The examples show that there is a conflict between allowing nondeterminism in specifications for P and Q, and preserving this liveness relationship under refinement. We have identified a notion of a plug-in relationship between P and Q, which requires that this liveness relationship is preserved by component-wise refinements.

The bicompositional listening and answering relations between P and Q defined above ensure that Q acts as a suitable plug-in to P. P nondeterministically triggers Q, which returns results back to P. If Q listens on T to P, and Q answers on R to P, for $\alpha(P) \cap \alpha(Q) = T \cup R$ then Q plugs-in to P , and if $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$ then Q' plugs-in t P. Thus we can confidently refine P and Q separately, knowing that refinements cooperate as desired.

7 Using the Definitions

The following small examples illustrate the use of the properties defined above. Components are defined using CSP.

Example 6. In the first case, P is a process which places a request with either event $a1$ or event $a2$ (chosen nondeterministically). Q is willing to accept either and provide an acceptable response, hence Q acts as a suitable plug-in to P.

$$P = (a1 \rightarrow r{-}{>}\, P) \sqcap (a2 \rightarrow r \rightarrow P)$$
$$Q = (a1 \rightarrow r \rightarrow Q) \;\Box\; (a2 \rightarrow r \rightarrow Q)$$

Q **listens on** $\{a1, a2\}$ **to** P – because Q has the same traces as P and refuses neither $a1$ nor $a2$ at any point where P might engage in them.
Q **answers on** $\{r\}$ **to** P – because Q is willing to provide a response whenever P expects one.

Example 7. We have a different situation if Q is able to deal with one of the possible requests only, rendering it an unsuitable plug-in to P.

$$P = (a1 \to r \to P) \sqcap (a2 \to r \to P) \qquad Q = (a1 \to r \to Q)$$

Q **does not listen on** $\{a1, a2\}$ **to** P – because $\langle a2 \rangle$ is a trace of P but $(\langle\rangle, \{a2\})$ is a failure of Q.

Example 8. The listening and answering relations are not inverses:

$$P = (a \to P) \sqcap STOP \qquad Q = (a \to Q)$$

Q **listens on** $\{a\}$ **to** P – because Q is willing to provide a response whenever P might expect one.
P **does not answer on** $\{a\}$ **to** Q – because $\langle a \rangle$ is a trace of Q but $(\langle\rangle, \{a\})$ is a failure of P.

Example 9. Here P requests a key which is subsequently supplied by Q. Q selects the value for the key which it outputs to P on channel *SupplyKey*. P is prepared to input any value on *SupplyKey*, hence, Q acts as a suitable plug-in to P.

$$P = RequestKey \to SupplyKey?k : KEY \to P$$
$$Q = RequestKey \to \sqcap_{k:KEY}(SupplyKey!k \to Q)$$

Q **listens on** $\{RequestKey\}$ **to** P – because Q is always ready to accept a key request whenever P issues one.
Q **answers on** $\{k : KEY \mid SupplyKey.k\}$ **to** P – because whenever P wants a key as input, it accepts any key offered, and Q is ready to offer one.

8 Using CSP Plug-Ins with Action Systems

The listening and answering relations we have identified are not completely general, but do typify a number of client-server behaviours. Their formulations in CSP failures models makes them meaningful for both Action Systems and CSP, providing a means of ensuring that two processes can be separately developed, whilst maintaining integrity of combined behaviour.

What remains is to to use the formal connection between the two notations to verify the properties for individually specified components. We are investigating practical approaches for verifying these properties for certain classes of "loosely-coupled" components illustrated by our secure database example.

For example, suppose a CSP specification for a key exchange protocol describes a process which is always willing to accept a request as input, and subsequently distribute a common key to the server and user, in nondeterministic order. It should be deliberately abstract to allow for a variety of specific key distribution protocols. This is to be viewed as a "minimum specification" of the key exchange to be refined and verified as a separate unit. We want to plug

this in to the Action System and show that the behaviour allowed by the CSP plug-in is acceptable to the database specification.

One of the advantages of relating components using bicompositional properties is that different components can be refined independently and the properties are guaranteed to hold between all refinements. There are various aspects of the CSP specification which we might want to further develop through refinement. One is to unfold specific details of a chosen key exchange protocol by describing an intended implementation, or an off-the-shelf component. This might introduce a trusted key server together with a prescribed sequence of events required by the protocol between user clients and the database server. We can verify that the implementation is valid by checking that, with additional events hidden, it is a refinement of Q, thus establishing that the chosen protocol behaves as expected.

A significant advantage of treating the security protocol as a suitable CSP plug-in is that we can naturally specify event-based behaviour and check relevant properties automatically using the FDR model checker, perhaps with various induction techniques (for example [CR99]) and data independence techniques (for example [Ros98,RL96]) for transcending bounded state. A significant disadvantage of using Action Systems for such aspects is that properly specifying allowable sequences of actions is very awkward. Another reason for refining the CSP specification is that we might want to analyse behaviour of a chosen protocol with respect to security, for example, robustness against deliberate or inadvertent attacks by intruders. For example, Lowe and Roscoe [LR97] discover potential security flaws with the TMN key exchange protocol, revealed by counter examples provided by FDR showing that attackers could perform operations specifically disallowed by the CSP specification.

9 Conclusion

The work described in this paper is the first step towards a general framework for describing and verifying the suitability of separate components working together to form an integrated system. Components are likely to be developed separately, using different notations and techniques. It is also likely that previously-developed components would be incorporated, in which case the component should provide an abstract specification (or specifications) as a minimum guarantee of its behaviour. In this way, a formal framework can be provided for larger systems created from a variety of loosely coupled subsystems, using off-the-shelf components where possible.

We have used relational constraints to characterise our notion of minimum requirements for a plug-in component: it must operate under the control of the main component in that the combination deadlocks only at the behest of the main. The approach also applies to independent processes which are not in a master-slave relationship but which are co-operating to achieve a one-off task. There may be other properties which would be useful to incorporate, perhaps tailored to the purposes of a specific system, but the notion of plug-in components which operate in accordance with the demands of a requesting module seems to be a generally useful one. Identifying bicompositionality between specifications underlines the fact that not all desirable relationships

between components would be preserved by refinement. Working with a property which is not bicompositional makes verification extremely difficult: either the property is proved at an early stage and continued effort is needed to check that the property is maintained as the system is refined, or the property is not checked till a later stage when proof is often more difficult and may reveal errors stemming from an early stage of development. The "listening" and "answering" properties provide sufficient conditions ensuring a bicompositional plug-in relationship. Another feature of these properties is that, unlike the formulation of the "as live as" relation, there is no need to directly deal with the parallel system of the two components. However, proving trace requirements is still not completely straightforward and efficient proof techniques are a topic of on-going research.

As sufficient conditions, we are aware that the properties given here rule out some cases which might otherwise be considered acceptable and that variations on the conditions might be devised to improve the situation. Other variations may be suitable in different circumstances. Further, there may be other useful paradigms characterising processes which could place constraints on input, such as insisting on receiving fresh keys. This is another area which we are continuing to develop. It is perhaps worth noting that in formulating bicompositional properties we encounter a difficulty similar to that of noninterference properties in security. Many different variants are possible and the effects are not always apparent until pathological examples are examined.

Another guiding principle has been the need to support different notations. The case study from which this work arose describes a secure database described using Action Systems which uses plug-in components for security services such as key exchange. The properties we have described are suitable for dealing with combinations of both CSP processes and Action Systems since traces and failures are defined for both. This allows both state-based and event-oriented specifications to be used for parts of the system to which they are best-suited and combined in a loosely-coupled manner. The approach is also applicable to other notations for which a traces/failures interpretation can be given. Again, the fact that the parallel system does not have to be calculated is an advantage. However, the identification of sub-traces within each component is still a significant task which forms part of the ongoing investigations referred to above.

As referred to in Section 3, work combining CSP with B is being undertaken by Butler [But99] and Treharne and Schneider [TS99,TS]. The main focus of these approaches has been to use CSP as a convenient way to specify constraints on the sequencing of, i.e., controlling the state-based actions. We take a different perspective, wishing to allow combinations of state-based and event-based components with either being designated as being "in control" as appropriate. However, the techniques developed for these approaches in which loop control requirements are "unwound" to give a set of verification conditions may be useful in establishing more easily provable conditions for our approach.

In common with Butler, Treharne and Schneider, our aim is to contain inherent problems of scale in applying formal techniques to large applications. The goal of a great deal of current research is to combine different formal approaches in order to treat different aspects of a given system. There is a

danger that combining techniques for a particular system creates prohibitive complexity. Our aim is to divide and conquer potential complexity by structuring specifications early in the development process, so that independent analysis can be effectively performed. Finally we note that our techniques are formulated using a traces/failures approach, but the concepts which we have identified are generally applicable to other formal and semi-formal specification techniques.

Acknowledgement. The authors would like to thank Mike Reed for his valuable discussions about various finer points of CSP semantics.

Appendix. A Taste of CSP

The CSP language is a means of describing components of systems, *processes* whose external actions are the communication or refusal of instantaneous atomic *events*. All the participants in an event must agree on its performance. We give a selection of CSP algebraic operators for constructing processes.

$STOP$ the simplest CSP process; it never engages in any action, never terminates.

$a \rightarrow P$ is the most basic program constructor. It waits to perform the event a and after this has occurred it behaves as process P. The same notation is used for outputs ($c!v \rightarrow P$) and inputs ($c?x \rightarrow P(x)$) of values on named channels.

$P \sqcap Q$ is *nondeterministic* or internal choice. It may behave as P or Q arbitrarily.

$P \Box Q$ is external or *deterministic* choice. It first offers the initial actions of both P and Q to its environment. Its subsequent behaviour is like P if the initial action chosen was possible only for P, and similarly for Q. If P and Q have common initial actions, its subsequent behaviour is nondeterministic (like $\sqcap$). A deterministic choice between $STOP$ and another process, $STOP \Box P$ is identical to P.

$P \parallel Q$ is parallel (concurrent) composition. P and Q evolve separately, but events in the intersection of their alphabets occur only when P and Q agree (i.e. *synchronise*) to perform them. (We use this restricted form of the parallel operator. The more general form allows processes to selectively synchronise on events.)

$P \,|||\, Q$ represents the interleaved parallel composition. P and Q evolve separately, and do not synchronise on their events.

Failures Model. The set of *failures* $\mathcal{F}$ satisfies the following axioms [BHR84].

$$(\langle\rangle, \{\}) \in \mathcal{F} \tag{M1}$$
$$(s \frown t) \in \mathcal{F} \Rightarrow (s, \{\}) \in \mathcal{F} \tag{M2}$$
$$(s, X) \in \mathcal{F} \wedge Y \subseteq X \Rightarrow (s, Y) \in \mathcal{F} \tag{M3}$$
$$(s, X) \in \mathcal{F} \wedge (\forall\, c \in Y \subseteq \alpha(P) \bullet ((s \frown \langle c\rangle, \{\}) \notin \mathcal{F} \Rightarrow (s, X \cup Y) \in \mathcal{F} \tag{M4}$$

To illustrate how CSP operations are defined with failures semantics, we give the failures for the parallel operator. We take the semantic view of having individual alphabets for each process [Hoa85] here to simplify the use of the parallel operator.

$$\begin{aligned}\mathcal{F}[\![P \parallel Q]\!] = \{(s, X \cup Y) \mid\ & x \in s \Rightarrow x \in \alpha(P) \cup \alpha(Q) \wedge \\ & (s \upharpoonright \alpha(P), X) \in \mathcal{F}[\![P]\!] \ \wedge \\ & (s \upharpoonright \alpha(Q), X) \in \mathcal{F}[\![Q]\!]\}\end{aligned}$$

References

[BHR84] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *JACM*, 31:560–599, 1984.

[BKS83] R.J.R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. In *2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 131–142, 1983.

[BR85] S.D. Brookes and A.W. Roscoe. An improved failures model for communicating sequential processes. In *Proc. Pittsburgh Seminar on Concurrency*. Springer, 1985.

[But92] M.J. Butler. *A CSP Approach to Action Systems*. DPhil thesis, University of Oxford, 1992.

[But99] M.J. Butler. csp2b: A practical approach to combining CSP and B. In J. Woodcock J. Davies, J.M. Wing, editor, *FM99 World Congress*. Springer Verlag, 1999.

[CR99] Sadie Creese and Joy Reed. Verifying end-to-end protocols using induction with CSP/FDR. In *Proc. of IPPS/SPDP Workshop on Parallel and Distributed Processing*, LNCS 1586, Lisbon, Portugal, 1999. Springer.

[For] Formal Systems (Europe) Ltd. *Failures Divergence Refinement.* User Manual and Tutorial*, version 2.11*.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. of TACAS*, volume 1055 of *LNCS*. Springer, 1996.

[LR97] G. Lowe and A.W. Roscoe. Using CSP to detect errors in the TMN protocol. *IEEE Trans. Soft. Eng.*, 23(10), 1997.

[Mea94] C. Meadows. The NRL protocol analyzer: An overview. *J. Logic Programming*, 19,20, 1994.

[Mor90] C.C. Morgan. Of wp and CSP. In D. Gries W.H.J. Feijen, A.G.M. van Gasteren and J. Misra, editors, *Beauty is our business: a birthday salute to Edsger W. Dijkstra*. Springer-Verlag, 1990.

[RL96] A.W. Roscoe and R.S. Lazic. Using logical relations for automated verification of data-independent CSP. In *Oxford Workshop on Automated Formal Methods ENTCS*, Oxford, UK, 1996.

[Ros98] A.W. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1998.

[RR99] G.M. Reed and A.W. Roscoe. The timed failures-stability model for CSP. *Theoretical Computer Science*, 211:85–127, 1999.

[RSG99] J.N. Reed, J.E. Sinclair, and F. Guigand. Deductive reasoning versus model checking: two formal approaches for system development. In K. Taguchi K. Araki, A. Galloway, editor, *Integrated Formal Methods 1999*, York, UK, June 1999. Springer Verlag.

[RSR99] J.N. Reed, J.E. Sinclair, and G.M. Reed. Routing - a challenge to formal methods. In *Proc. of International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, 1999. CSREA Press.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd ed., 1992.

[TS] H. Treharne and S. Schneider. How to drive a B machine. To appear.

[TS99] H. Treharne and Schneider S. Using a process algebra to control B operations. In K. Taguchi K. Araki, A. Galloway, editor, *Integated Formal Methods*, pages 437–456, York, UK, 1999. Springer Verlag.

[WM90] J.C.P. Woodcock and C.C. Morgan. Refinement of state-based concurrent systems. In *Proc. of VDM Symposium*, LNCS 428. Springer-Verlag, 1990.

Proving Deadlock Freedom in Component-Based Programming*

Paola Inverardi[1] and Sebastian Uchitel[2]

[1] Dip. di Matematica, Universit'a dell' Aquila, I-67010, L'Aquila, Italy. email: inverard@univaq.it
[2] Dep. of Computing, Imperial College, Queen's Gate 180, London, SW7 2BZ, UK. email: su2@doc.ic.ac.uk

Abstract. Emerging technologies such as commercial off-the-shelf products (COTS) and component integration frameworks such as CORBA and COM are changing the way software is produced. Distributed applications are being designed as sets of autonomous, decoupled components, allowing rapid development based on integration of COTS and simplifying architectural changes required to cope with the dynamics of the underlying environment. Although integration technologies and development techniques assume rather simple architectural contexts, they face a critical problem: Component integration.
So far existing techniques for detecting dynamic integration errors are based on behavioural analysis of the composed system and have serious space complexity problems. In this work we propose a broader notion of component semantics based on assumptions and a method for proving deadlock freedom in a component-based setting. Our goal is to prevent and detect these errors in component based programming settings in a component-wise fashion. We aim for effective methods that can scale to real size applications even at the price of incompleteness as opposed to many existing methods that although theoretically complete might fail in practice.

1 Introduction

In recent years important changes have taken place in the way we produce software artefacts. On one side, software production is becoming more involved with distributed applications running on heterogeneous networks. On the other, emerging technologies such as commercial off-the-shelf (COTS) products are becoming a market reality for rapid and cheap system development [14]. Although these trends may seem independent, they actually have been bound together with the wide spreading of component integration technologies such as CORBA and COM. Distributed applications are being designed as sets of autonomous, decoupled components, allowing rapid development based on integration of COTS

* P. Inverardi was artially supported by the Italian MURST national Project SALADIN. S. Uchitel was partially supported by ARTE Project, PIC 11-00000-01856, ANPCyT and TW72, UBACyT.

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 60–75, 2001.

and simplifying architectural changes required to cope with the dynamics of the underlying environment. Integration technologies and development techniques assume rather simple architectural contexts, usually distributed, with simple interaction capabilities. Nevertheless they face critical problems that pose a challenging research issues. For example, consider this quote from a recent US Defence Department briefing:

> *"A major theme of this year's demonstrations is the ability to build software systems by composing components, and do it reliably and predictably. We want to use the right components to do the job. We want to put them together so the system doesn't deadlock."*[1]

While for type integration and interface checking, type and sub-typing theories play an important role in preventing and detecting some integration errors, interaction properties remain problematic. Component assembling can result in architectural mismatches when trying to integrate components with incompatible interaction behaviour (e.g. [5]), resulting in system deadlocks, livelocks or failing to satisfy desired general functional and non-functional system properties. So far existing techniques for detecting dynamic integration errors are based on behavioural analysis (e.g. [6,4]) of the composed system model. The analysis is carried on at system level, possibly in a compositional fashion [6] and has serious problems with state explosion. Our goal is to prevent and detect these errors in component based programming settings in a component-wise fashion. We aim for effective methods that can scale to real size applications even at the price of incompleteness as opposed to many existing methods that although theoretically complete might fail in practice.

Our approach exploits the standardization and simplicity of the interaction mechanisms present in the component-based frameworks. We overcome the state explosion problem in deadlock verification for a significant number of cases. Our approach is based on enriching component semantics with additional information and performing analysis at a component level without building the system model. We start off with a set of components to be integrated, a composition mechanism, in this case full synchronization, and a property to be verified, namely deadlock freedom. We represent each component with an ACtual behaviour (AC) graph. An ASsumption (AS) graph for proving deadlock freedom is derived from each AC graph. Our checking algorithm processes all AC and AS graphs trying to verify if the AC graphs provide the requirements modelled by all the AS graphs. The algorithm works by finding pairs of AC and AS graphs that match through a suitable partial equivalence relation. According to the match found, arcs of the AS graph that have been provided for (covered arcs) are marked, and root nodes of both AC and AS graphs are updated. The algorithm repeats this process until all arcs of all AS graphs have been covered or no matching pair of graphs can be found. The former implies deadlock freedom of the system while the latter means that the algorithm cannot prove system deadlock freedom. Consequently,

[1] http://www.dyncorp-is.com/darpa/meetings/edcs99jun/

our algorithm is not complete (there are deadlock free systems that the algorithm fails to recognize), which is the price we must pay for tractability.

Summarizing, the contributions of this work are a broader notion of component semantics based on assumptions and a method for proving deadlock freedom in a component-based setting that is very efficient in terms of space-complexity. While the space complexity of our approach is polynomial, existing approaches have exponential orders of magnitude.

In the next section we discuss related work. In Section 3 we informally introduce the characteristics of a simple component/configuration language based on CCS and recall the definition of Labelled Transition Systems which are our basic model. In Section 4 we illustrate a simple case study that is used in Section 5 to present our approach. We discuss the methods completeness and complexity and conclude with final comments and future work.

2 Related Work

In order to obtain efficient verification mechanisms in terms of space complexity, there has been much effort to avoid the state explosion problem. There are two approaches: compositional verification and minimization. The first class verifies properties of individual components and properties of the global system are deduced from these (e.g. [12]). However as stated in [12] when verifying properties of components it may also be necessary to make assumptions about the environment, and the size of these assumptions is not fixed. Our approach shares the same motivation but it verifies properties of the component context using fixed size AS graphs. The compositional minimization approach is based on constructing a minimal semantically equivalent representation of the global system. This is managed by successive refinements and use of constraints and interface specifications [6,7]. However, these approaches still construct some kind of global system representation, therefore are subject to state explosion in worst cases. Neither efficient data representations such as Binary Decision Diagrams [3] nor most recent results like [1] have solved the space complexity problem.

From the perspective of property checking in large software systems, work in the area of module interconnection and software architecture languages can be mentioned, however the focus is not on efficient property verification of dynamic properties nor is the specific setting of component-based programming taken into account. For an extensive treatment of this aspect refer to [9].

There have been other attempts at proving deadlock freedom statically. Interesting results in this direction can be found in [10] where a type system is proposed that ensures (certain kinds of) deadlock freedom through static checking. The approach is based on including the order of channel use in the type information and requiring the designer to annotate communication channels as reliable or unreliable. As in our work, they use behavioural information to enhance the type system, however part of the additional information must be provided by users and is related to channels rather than components. In our approach, additional information is derived from the property to be proved and

the communication context. Besides, the derived information extends component semantics, thus integrating well with the current direction that software development has taken, based on component integration technologies and commercial off-the-shelf products. In [2] component behaviour is decomposed into interface descriptions, which then can be used to prove deadlock freedom. The method is more incomplete than ours as it requires components not to exhibit non-deterministic behaviour that can be resolved by the influence of their environment. In other words, a non-deterministic choice involving a component input is not allowed. Our approach allows this kind of non-determinism, furthermore, the example used in this paper exhibits many of these non-determinisms.

The method presented in this paper originates from the work in [9]. However it differs in a number of ways:

- The present method is more complete. This is mainly because of the partial equivalence relation used in this approach: We do not require the whole behaviour of a component to be used when providing a portion of another component's assumption. Relaxing this requirement allows more systems to be checked. Detailed examples can be found in [8].
- The component/configuration language is well founded and simpler.
- There is a clear distinction between assumption generation and assumption checking.
- There is a clear distinction between notions of equivalence and partial matching, therefore it is possible to adapt the definitions for other notions of equivalence, allowing for example to switch from synchronous to asynchronous communication.

3 A Basic Component-Configuration Language

Our model for component-based systems describes components in terms of their input and output actions using labelled transition systems (LTS) (Definition 1). Input and outputs are considered to be blocking actions, thus we shall work with the (synchronous) parallel composition of LTS. System description using LTS and parallel composition is widely used in research (e.g. [6,4]) and we have chosen CCS [11] as our specification language mainly for its simplicity and firm foundations. Thus, in this work component behaviour shall be described as CCS processes and system configuration shall be specified using the parallel composition and restriction operators. As LTSs of all examples used in this paper are shown, knowledge of CCS is not critical to follow the main ideas of this paper.

Definition 1 (Labelled Transition Systems). *A component C is modelled by a labelled transition system $< S, L, \rightarrow, s >$, where S is a set of states; s is the initial component state; L is a set of labels representing the channels through which the component can communicate; $\rightarrow \subseteq (S \times Act \times S)$ is a transition relation that describes the behaviour of the component.*

4 The Compressing Proxy Problem

In this section we briefly present the Compressing Proxy example. For a more detailed explanation refer to [9]. To improve the performance of UNIX-based World Wide Web browsers over slow networks, one could create an HTTP (Hyper Text Transfer Protocol) server that compresses and uncompresses data that it sends across the network. This is the purpose of the Compressing Proxy, which weds the **gzip** compression/decompression program to the standard HTTP server available from CERN.

The main difficulty that arises in the Compressing Proxy system is the correct integration of existing components. The CERN HTTP server consists of *filters* strung together in series executing in one single process, while the gzip program runs in a separate UNIX process. Therefore an adaptor must be created to coordinate these components correctly (see Figure 1).

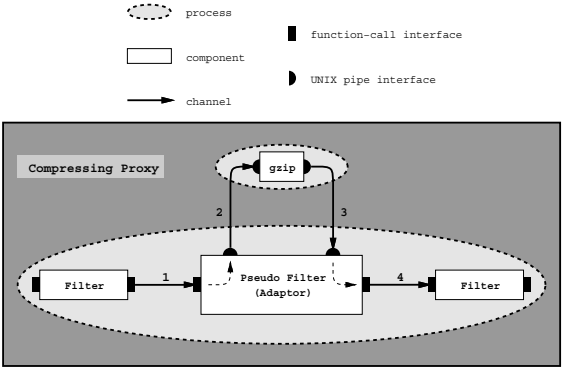


Fig. 1. The Compressing Proxy

However the correct construction of the adaptor requires a deep understanding of the other components. Suppose the adaptor simply passes data on to gzip whenever it receives data from the upstream filter. Once the stream is closed by the upstream filter (i.e., there are no more data to be compressed), the adaptor reads the compressed data from gzip and pushes the data toward the downstream filter. At a component level, this behaviour makes sense. But at a global system level we can experience deadlock.

In particular, gzip uses a one-pass compression algorithm and may attempt to write a portion of the compressed data (perhaps because an internal buffer is full) before the adaptor is ready, thus blocking. With gzip blocked, the adaptor also becomes blocked when it attempts to pass on more of the data to gzip, leaving the system in deadlock.

A way to avoid deadlock in this situation is to have the adaptor handle the data incrementally and use non-blocking reads and writes. This would allow the

adaptor to read some data from gzip when its attempt to write data to gzip is blocked.

We model all four system component behaviours, gzip, Adaptor, Upstream and Downstream CERN filters as the CCS processes in Table 4. The Upstream CERN filter is very simple, it can continuously perform output actions through its *upstream* interface point (u). Similarly, the Downstream Filter can perform input actions through its *downstream* port (d). The gzip interface consists of a port for inputting the source file (s), one for outputting the compressed file (z), and two other ports to model *end of source file* (es) and *end of compressed file* (ez). Finally the adaptor interacts with all the other components, and therefore its interface is the union of all the other component interfaces. The complete system configuration is given by (UF | GZ | AD | DN) $\backslash\{u, s, es, z, ez, d\}$ and the following CCS processes:

Upstream Filter (UF)	**Downstream Filter (DF)**
UF $\stackrel{def}{=} \overline{u}$.UF	DF $\stackrel{def}{=} d$.DF
GZip (GZ)	**Adaptor (AD)**
GZ $\stackrel{def}{=} s$.In	AD $\stackrel{def}{=} u.\overline{s}$.ToGZ
In $\stackrel{def}{=} s$.In + $es.\overline{z}$.Out + $\tau.\overline{z}$.Out	ToGZ $\stackrel{def}{=} \overline{s}$.ToGZ + $\overline{es}.z$.FromGZ
Out $\stackrel{def}{=} \overline{z}$.Out + $\overline{ez}$.GZ + τ.GZ	FromGZ $\stackrel{def}{=} z$.FromGZ + $ez.\overline{d}$.AD

5 Property Checking Using Assumptions

We represent component behaviour (and component assumptions later on) with directed, rooted graphs that simply extend labelled transition systems to allow multiple root nodes:

Definition 2 (Graphs). *A (directed rooted) graph G is a tuple of the form (N_G, L_G, A_G, R_G) where N_G is a set of nodes, L_G is a set of labels with $\tau \in L_G$, $A_G \subseteq N_G \times L_G \times N_G$ is a set of arcs and $R_G \subseteq N_G$ is a nonempty set of root nodes.*

- *We shall write $g \stackrel{l}{\rightarrow} h$, if there is an arc $(g, l, h) \in A_G$. We shall also write $g \rightarrow h$, meaning that $g \stackrel{l}{\rightarrow} h$ for some $l \in L_G$.*
- *If $t = l_1 \cdots l_n \in {L_G}^*$, then we write $g \stackrel{t}{\longrightarrow}^* h$, if $g \stackrel{l_1}{\rightarrow} \cdots \stackrel{l_n}{\rightarrow} h$. We shall also write $g \longrightarrow^* h$, meaning that $g \stackrel{t}{\longrightarrow}^* h$ for some $t \in {L_G}^*$.*
- *We shall write $g \stackrel{l}{\Rightarrow} h$, if $g \stackrel{t}{\longrightarrow}^* h$ for some $t \in \tau^*.l.\tau^*$.*

We define the notion of Actual Behaviour (AC) Graph for modelling component behaviour. The term actual emphasizes the difference between component behaviour and the intended, or assumed, behaviour of the environment. AC

graphs model components in an intuitive way. Each node represents a state of the component and the root node represents its initial state. Each arc represents the possible transition into a new state where the transition label is the action performed by the component.

Definition 3 (AC Graphs). *Let $< S, L, \rightarrow, s >$ be a labelled transition system for component C. We call a graph of the form $(S, L, \rightarrow, \{s\})$ the ACtual behaviour graph (AC graph) of a component C. We shall usually denote nodes in AC with ν.*

In the rest of the section we will show how component assumptions can be derived and used for proving deadlock freedom in a system composed of a finite number of components that communicate synchronously. Following a common hypothesis in automated checking of properties of complex systems [6], behaviour of all components can be finitely represented. In addition, in order to simplify presentation, we add two constraints:

- Components can perform each computation infinitely often. In other words, all nodes of a components AC graph are reachable from any other node. This condition simplifies the presentation of the partial equivalence relation. It can easily be dropped by introducing a proper treatment of *ending* nodes in the definitions below.
- There are no "shared" actions. This means that every communication is only used by two components. Again, this condition simplifies the presentation of the checking algorithm and can be easily dropped at the expense of more checks.

It is worth noticing that dropping the first condition has no implication on the complexity results of this presentation, while for the second only time complexity changes. In Section 6 we discuss more deeply the implications of both constraints.

5.1 Deriving Assumptions for Deadlock Freedom

We wish to derive from the behaviour of a component the requirements on its environment that guarantee deadlock freedom. A system is in deadlock when it cannot perform any computation, thus in our setting, deadlock means that all components are blocked waiting for an action from the environment that is not possible. Our approach is to verify that no components under any circumstance will block. This conservative approach suffices to prove deadlock freedom exclusively from component assumptions. The payback, as we shall show, is efficiency, while the drawback is incompleteness.

As components are combined together composing them in parallel and restricting all actions, a component will not block if its environment can always provide the actions it requires for changing state. Thus we can define the notion of component assumption in the context of parallel composition and deadlock freedom in the following way:

Definition 4 (AS Graphs). *Let (N, L, A, R) be an AC graph of a component C, then the corresponding ASsumption (AS) graph is (N, L, A', R) where $A' = \{(\nu, \overline{a}, \nu') \mid (\nu, a, \nu') \in A\}$. We shall usually denote nodes in AS graphs with μ.*

The AC and AS graphs for the components of the Compressing Proxy appear in Figure 2.

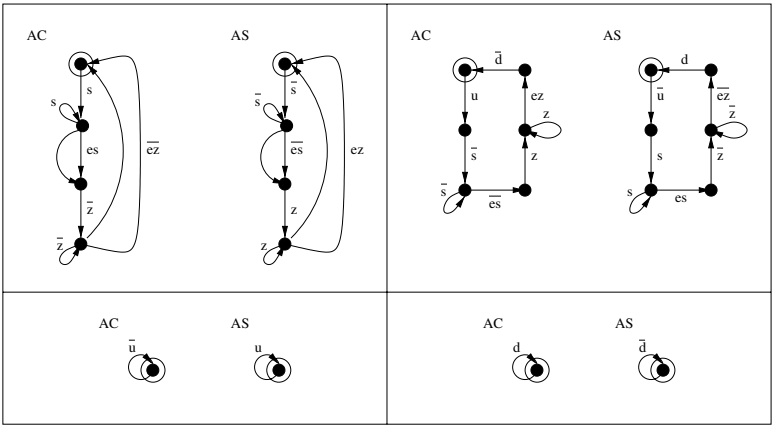


Fig. 2. Graphs for components GZ, AD, UF, SF (starting top-left)

5.2 Checking Assumptions

Once component assumptions have been derived, we wish to verify if the environment satisfies these assumptions. The environment corresponds to the rest of the components in the given context. This satisfaction relation reduces to proving if the component environment is equivalent to the component assumption with the following notion of equivalence:

Definition 5 (Equivalence). *Let G and H be graphs with the same alphabet, i.e. $L_G = L_H$. We define $\approx$ to be the union of all relations ρ where $R_G \times R_H \subseteq \rho$ and if $(g, h) \in \rho$ the following two conditions hold:*

- $g \xrightarrow{l}_G g' \Rightarrow (\exists\, h' : h \overset{l}{\Rightarrow}_H h' \wedge (g', h') \in \rho) \wedge (\forall\, h' : h \overset{l}{\Rightarrow}_H h' \Rightarrow (g', h') \in \rho)$, *and*
- $h \xrightarrow{l}_H h' \Rightarrow (\exists\, g' : g \overset{l}{\Rightarrow}_G g' \wedge (g', h') \in \rho) \wedge (\forall\, g' : g \overset{l}{\Rightarrow}_G g' \Rightarrow (g', h') \in \rho)$.

The idea behind the definition of equivalence is that the graphs can always imitate each other. If a graph performs an action l, the other graph can also perform l and, no matter what internal choices it may make, it will be able to continue imitating the other graph. Note that our notion of equivalence is more restrictive than the notion of weak bisimilarity [11] since we need to assure that a

given behaviour *must* be provided by all the branches that provide the matched portion. This is what, in the above clauses, the for-all conditions express.

We verify the equivalences between AS graphs and environments without constructing the whole environment behaviour. The main idea is to allow a portion of component behaviour to provide a portion of another component assumption. For this we need to provide a notion of *partial equivalence* that preserves equivalence in a conservative way (Section 5.2). Once a partial equivalence has been established, the assumption graph has been satisfied to some extent and therefore some marking mechanism is necessary in order to record it. The checking algorithm of Section 5.2 iteratively finds partial equivalences and marks the assumptions accordingly until all assumptions have been satisfied.

Partial Equivalence. A partial equivalence between an AC and AS graph allows the equivalence relation to be defined up to a certain point in the graphs. The AC and AS graphs are not required to be completely equivalent, their root nodes must be equivalent, the nodes reachable from root nodes too, and so on until set of nodes called stopping nodes is reached. Stopping nodes represent the points where the actual behaviour will stop providing the assumption's requirements, hence there should be another AC graph capable of doing so from then on. We now formally introduce these notions. The notion of stopping nodes is needed to guarantee that the graph portions included in the partial equivalence correspond to behaviours starting from root nodes onwards. Throughout the following definitions, given a relation ρ and an element a, we shall write $a \in \rho$ if there is an element b such that $(a, b) \in \rho$ or $(b, a) \in \rho$.

Definition 6 (Stopping Nodes). *Let G and H be graphs and ρ a relation in $N_G \times N_H$. We say that the set $S_{G,\rho} = \{g \mid g \in N_G \wedge g \in \rho \wedge g \not\in R_G \wedge (g \xrightarrow{l} g' \Rightarrow (g' \not\in \rho \vee g' \in R_G))\}$ is the set of stopping nodes of ρ in G . We omit the symmetric definition for $S_{H,\rho}$.*

Informally a stopping node is a node in the relation ρ that is not a root node and for which no other nodes in ρ are reachable.

Definition 7 (Partial Equivalence). *Let G_{ac} be an actual behaviour graph with nodes ν_i, G_{as} be an assumption graph with nodes μ_j. We define $\simeq$ to be the union of all relations ρ such that $R_{G_{ac}} \times R_{G_{as}} \subseteq \rho$ and all the following hold:*

1. *if $\nu \in \rho$ then $\nu \in R_{G_{ac}}$ or there is a node $\nu' \in \rho$ such that $\nu' \rightarrow \nu$.*
2. *if $(\nu, \mu) \in \rho$, $\nu \not\in S_{G_{ac},\rho}$ and $\mu \not\in S_{G_{as},\rho}$ then*
   a) $\nu \xrightarrow{l} \nu' \Rightarrow (\exists\, \mu' : \mu \stackrel{l}{\Rightarrow} \mu' \wedge (\nu', \mu') \in \rho) \wedge (\forall\, \mu' : \mu' \stackrel{l}{\Rightarrow} \mu \Rightarrow (\nu', \mu') \in \rho)$, *and*
   b) $\mu \xrightarrow{l} \mu' \Rightarrow (\exists\, \nu' : \nu \stackrel{l}{\Rightarrow} \nu' \wedge\ (\nu', \mu') \in \rho) \wedge (\forall\, \nu' : \nu' \stackrel{l}{\Rightarrow} \nu \Rightarrow (\nu', \mu') \in \rho)$.
3. *if $(\nu, \mu) \in \rho$, $\nu \in S_{G_{ac},\rho}$ or $\mu \in S_{G_{as},\rho}$ then*
   a) *if $\nu \in R_{G_{ac}}$ then $(\nu' \xrightarrow{l} \nu \Rightarrow \exists\, \mu' : \mu' \stackrel{l}{\Rightarrow} \mu)$, and*
   b) *if $\mu \in R_{G_{as}}$ then $(\mu' \xrightarrow{l} \mu \Rightarrow \exists\, \nu' : \nu' \stackrel{l}{\Rightarrow} \nu)$.*

4. *if* $(\nu, \mu) \in \rho$ *and* $\nu \in S_{G_{ac},\rho}$ *then* $\mu \in S_{G_{as},\rho}$ *or* $\mu \in R_{G_{as}}$.
5. *if* $(\nu, \mu) \in \rho$ *and* $\mu \in S_{G_{as},\rho}$ *then* $\nu \in S_{G_{ac},\rho}$ *or* $\nu \in R_{G_{ac}}$.

We say that G_{ac} *and* G_{as} *are partially equivalent if* $G_{ac} \simeq G_{as}$.

The definition obviously resembles the definition of equivalence of Def. 5. We succinctly explain the additions: A partial equivalence relation does not necessarily cover all nodes of each graph, thus Rule 1 is needed to enforce that the nodes that are included in the relation are connected. By Rule 2, all nodes that are related and do not belong to the boundary where the actual behaviour stops providing the assumption requirement, are required to be equivalent (in the same sense as in Definition 5). Rules 3, 4 and 5 involve stopping nodes, the main idea is that equivalence is not required from these nodes onwards. However, stopping nodes must be related to stopping nodes, or if there is a loop (i.e. a path that returns to a root node) in one graph, a stopping node might be related to a root node (Rules 4 and 5). Rule 3 deals with spurious pairs of stopping and root nodes, by requiring the stopping node to represent the end of looping paths in the other graph. In short, all rules but 3 are intended for the *partial* part of the definition while Rule 3 is intended for the *equivalence* part.

Checking Algorithm. The checking algorithm is very simple, it iteratively finds partial equivalences between AC and AS graphs, marks all the fulfilled assumptions and changes the roots of both graphs. Iteration stops when all assumptions are completely marked. An important point is that partial equivalences guarantee that the matched portions of assumptions cannot be matched in any other way, therefore the order in which partial matches are applied does not affect the correctness of the algorithm.

The checking algorithm that is presented below intends to clarify how the checking of component assumptions is done and to provide a basis for correctness, completeness and complexity. By no means is the algorithm, optimum. Many heuristics could be built into it in order to increase time and space efficiency, however, at this stage of our research we are interested in orders of complexity.

Definition 8 (Covered Arcs). *Let* G_{ac} *be an actual behaviour graph,* G_{as} *be an assumption graph and* $\simeq$ *a partial equivalence relation such that* $G_{ac} \simeq G_{as}$, *then we say that an arc* $(\mu, l, \mu') \in A_{G_{as}}$ *is covered if* $\mu, \mu' \in \simeq$ *and* $\mu \not\in S_{G_{as},\simeq}$.

Definition 9 (Checking Algorithm). *Let* $\Gamma_{ac} = \{G_{ac_1}, G_{ac_2}, \ldots, G_{ac_n}\}$ *be a set of AC graphs and* $\Gamma_{as} = \{G_{as_1}, G_{as_2}, \ldots, G_{as_n}\}$ *the set of corresponding AS graphs.*

1. *Let* $G'_{ac_i} = G_{ac_i}$ *for every* $G_{ac_i} \in \Gamma_{ac}$.
2. *If* Γ_{as} *is empty then*
   - *If* $G'_{ac_i} \approx G_{ac_i}$ *for every* $G_{ac_i} \in \Gamma_{ac}$, *return true.*
   - *Otherwise return false.*

3. *Try to find an AC graph G_{ac_i} in Γ_{ac}, an AS graph G_{as_j} in Γ_{as}, and a partial equivalence between them ($G_{ac_i} \simeq G_{as_j}$). If it is not found, return false.*
4. *Relabel with τ every arc in G_{as_j} that is covered by $\simeq$. If all arcs in G_{as_j} are labeled τ remove it from Γ_{as}.*
5. *If $S_{as} \cup S_{ac} \not= \emptyset$ then let $R_{G_{as_i}} = \{\mu \mid \mu \in S_{as} \vee \exists \nu \in S_{ac} : \nu \simeq \mu\}$ and let $R_{G_{ac_i}} = \{\nu \mid \nu \in S_{ac} \vee \exists \mu \in S_{as} : \nu \simeq \mu\}$.*
6. *Go to step 2.*

Note that the algorithm returns true only if all assumptions have been satisfied and the system configuration is in a setting equivalent to its initial state (Step 4).

We now apply the algorithm to the Compressing Proxy example. We represent partial equivalences with dotted lines for related nodes and crosses for stopping nodes. In figure 3, the Upstream Filter matches successfully with the Adaptor. Once the successful match has been made, both graphs are modified. The new state of the Adaptor can be seen in figure 4.

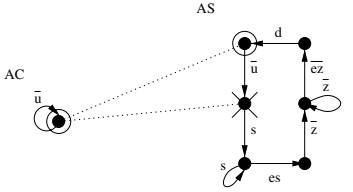


Fig. 3. Successful match of Upstream Filter AC and Adaptor AS graphs

Figure 4 shows how a partial match can be established between the gzip AC graph and the Adaptor AS graph. However it is possible to see that there is no way of extending the relation in order to cover the edge labelled *es*. Hence the algorithm, after all possible attempts, terminates at Step 2 returning false; meaning that the proposed configuration is presumably not deadlock free.

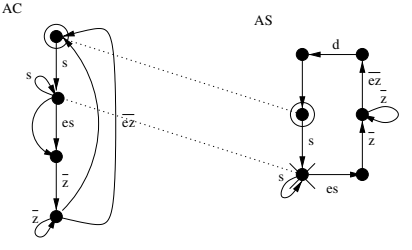


Fig. 4. Unsuccessful match of gzip AC and Adaptor AS graphs

Notice that the mismatch occurs precisely where the deadlock in the system appears: The gzip may attempt to start outputting the gzipped file ($\overline{z}$) while the adaptor is expecting to be synchronizing with a component inputting an *end of source* (es) before the gzipped file is outputted.

As mentioned in Section 4, the adaptor must be modified to prevent system deadlock. We propose a new Adaptor component and show that the algorithm proves the new system is deadlock free.

New Adaptor (NAD)

$\text{NAD} \stackrel{def}{=} u.\overline{s}.\text{ToGZ}$
$\text{ToGZ} \stackrel{def}{=} \overline{s}.\text{ToGZ} + \overline{es}.z.\text{FromGZ} + \tau.z.\text{FromGZ}$
$\text{FromGZ} \stackrel{def}{=} z.\text{FromGZ} + ez.\overline{d}.\text{NAD} + \tau.\overline{s}.\text{ToGZ}$

In figure 5, the partial equivalence that covers the $\overline{es}$ edge allows the New Adaptor AS graph to be updated, and Figure 6 finishes covering the AS graph completely. The algorithm goes on matching AC and AS graphs until all arcs of all AS graphs are covered. Thus the checking algorithm finally returns true, meaning that the proposed system is deadlock free.

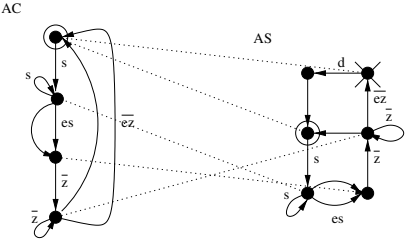


Fig. 5. Successful match of gzip AC and New Adaptor AS graphs

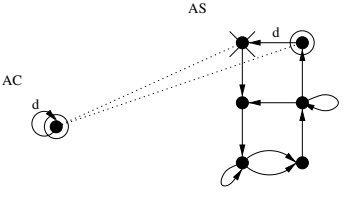


Fig. 6. Matching Downstream Filter AC and New Adaptor AS graphs

6 Method Assessment

Up to now, we have presented a method for checking deadlock freedom that trades off completeness for efficiency. We now show some evidence to substantiate the positive characteristics of our approach compared to existing ones. Because our work is in an initial stage, we cannot give, yet, empirical evidence, however a theoretical assessment on the completeness and complexity of our approach reveals that an implementation will yield good results.

Complexity. The algorithm presented above offers a partial solution to the state explosion problem. In our approach, deadlock freedom is proven without building the entire finite-state model of the system. We only construct finite representations of each component individually: an actual behaviour graph and an assumed behaviour graph of its context.

In standard approaches, using reachability analysis, the complete state space of the system is built. If we consider a concurrent system composed of N components of comparable size, whose finite state representation is of size $O(K)$, then the composed system state space is $O(K^N)$. Although there are many techniques for reducing the state space, such as automata minimization and "on the fly" algorithms, worst case still requires the whole state space to be analysed, leading to a time complexity of $O(K^N)$.

In our approach only two copies of each component are built, AC and AS graphs, thus following the same considerations as before, the state space complexity is radically improved to $O(KN)$. On the other hand, in terms of time complexity, the worst case of our algorithm is $O(N^3K^4log(K))$, which is comparable to the worst case of standard reachability. The time complexity results from the following: Establishing a partial equivalence relation between two graphs can be considered a variation of the standard bisimulation checking, thus its complexity would be upper bounded by $O(K^2log(K))$ [13]. However, the partial equivalence must be established for a pair of graphs, thus all possible pairs must be checked ($Comb(N, 2)$), leading us to $O(N^2(K^2log(K)))$. Finally, considering the worst case in which each partial match only covers a single arc of the NK^2 possibilities, we get $O(K^2N^3(K^2log(K)))$ which reduces to $O(N^3K^4log(K))$.

Completeness. Completeness is an important issue in our approach. We have mentioned that our method is not complete and in this section we discuss how incomplete it is and possible improvements. With respect to correctness, the proof is sketched in [8].

There are two different sources of incompleteness in our approach. Firstly, because at the beginning of Section 5 we constrained the systems for which the method can be used. Secondly, because our checking algorithm may not be able to conclude deadlock freedom for some deadlock free systems.

The first restriction of Section 5 that requires components to be able to perform each computation an infinite number of times does not affect the completeness of our approach. The goal of this restriction is to simplify the presentation

of definitions. Dropping the constraint requires defining the concept of recursive arcs as arcs that can be taken infinite times and modifying the matching scheme: recursive arcs of AS graphs can only be matched with recursive AC graphs while non-recursive arcs must match with non-recursive arcs. Definitions of this kind can be found in [9].

The restriction on shared channels is more serious. If a channel can be used by more than two components, there is potential *global* non-determinism in the overall system behaviour. A component may have the possibility of synchronizing with one of several components leading to a non-deterministic choice on the partner to which synchronize with. In terms of our approach, this means that one cannot commit to which AC graph will provide the AS graph requirements. Because the matching process guarantees that the matched arcs of the AS graph will always be provided by the AC graph, no matching can be done. This non-determinism introduced by shared channels is similar to the non-determinism that makes our algorithm incomplete; therefore we will discuss a solution to both problems farther on in the section.

Having discussed the restriction imposed on components, the incompleteness of the checking algorithm remains. Our approach is intrinsically incomplete. We attempt to prove a global property such as deadlock freedom in terms of local properties of each component. That is we say that the system is deadlock free if no system component can ever block. Obviously this is a strong sufficient condition but by no mean a necessary one.

As a consequence, the characteristics of our setting lead to the following situation: Given a deadlock free system, the algorithm may not be able to conclude that it is deadlock free. The algorithm reaches a state in which it cannot do further matches between AC and AS graphs. One reason for this is that the definition of partial match may be just too restrictive, leaving out some matches that might be correct. Compared to the previous version of this approach [9] the notion of partial match represents an improvement since it does not require matching between AC and AS graphs to use the entire AC graph to prove deadlock freedom. Examples that can be verified with the partial matching introduced in this paper and not with the previous versions of the approach can be found in [8].

However, the main reason for the incompleteness of our approach is connected to non-determinism. When there is a non-deterministic choice in component behaviour, when a component can interact with one of two different components, there can be not a unique matching that guarantees how the system will evolve. In these situations the algorithm stops without obtaining AS graphs completely matched, and therefore not giving a conclusive answer (Note that this is still less restrictive than the approach in [2]). In order to solve this problem we are working in the direction of changing the main algorithm in the following way: When there is a choice in the AS graph and it is not possible to match all choices simultaneously, the choices must be selected in turns. For each selection, the algorithm proceeds matching. If the checking algorithm succeeds matching all AS graphs, it tries the rest of the choices. If checking succeeds for all choices,

then it succeeds for the complete AS graph. This improvement on completeness does not have a drastic impact on complexity. In terms of time complexity, there is an important increase; however, in terms of space complexity, the modification presents no significant changes. No new graphs must be represented, the algorithm has to register the non-deterministic points and go back to them. Thus, space complexity will not jump to an exponential order, maintaining the advantages the approach has with respect to other methods.

Summarizing, the approach we present is incomplete but we think there is room for significant improvements. On the other hand, incompleteness is the price that must be paid to make analysis tractable. Our method may apply only to a subset of problems but it lowers complexity of the solution from exponential order to a polynomial one.

7 Conclusions and Future Work

In this work we have presented a broader notion of component semantics based on assumptions and a derived space-efficient method for proving deadlock freedom in a component based setting. This method is based on deriving assumptions (component requirements on its environment in order to guarantee a certain property in a specific composition context) and checking that all assumptions are guaranteed through a partial matching mechanism. The method is considerably more efficient than methods based on system model behaviour analysis, its space complexity is polynomial while existing approaches have exponential orders of magnitude. It is not complete but it allows the treatment of systems whose synchronization patterns are not trivial. We think it can be a useful tool to be included in a verification tool-set together with complete but not always applicable ones. Our approach heavily relies on the component-based setting. This is a very interesting context in which experimenting new verification techniques. In fact, on one side components by definition force standardization and therefore simplifications of the integration frameworks. On the other side there is room for suggestions on the kind of information that a component should explicitly carry on with it in order to be integrated in all suitable contexts. That is the notion of component semantics is conceived in broader terms than in traditional programming.

Dynamic properties are difficult to be proven and, as we discussed in Section 2, most of the proposed approaches to overcome state explosion are based on characterizing local properties and then try to ensure that these properties can be lifted up to the global system level. Our contribution is actually in this line, but with the aim of fixing once and for all the kind of information that a component has to carry with it independently of the contexts it will eventually be used. To this respect, we believe that assumptions are a good way to extend component semantics in order to verify properties more efficiently.

Ongoing and future work goes in several directions. Firstly, we are working on the validation of the framework through experimental results. We are currently working on an implementation of the algorithm, and considering other coordi-

nation contexts like non-fully synchronized or asynchronous ones. In particular we are trying to cast our approach in given architectural styles and experiment with commercial component base frameworks as COM. Second, we wish to extend the approach to deal with other properties such as general liveness and safety properties. To this respect we are thinking of general safety properties expressed with property automata that may be decomposed into component assumptions or specific component assumptions such as particular access protocols for shared resources. Lastly, we are working on an extension of the algorithm in order to improve completeness of our approach.

Acknowledgements. We would like to thank Alexander Wolf and Daniel Yankelevich for discussions and previous work on the subject of the paper.

References

1. R. Alur, L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang. Automatic modular verification. In *Proceedings of CONCUR '99: Concurrency Theory*, 1999.
2. F. Arbab, F.S. de Boer, and M. M. Bonsangue. A logical interface description language for components. In COORDINATION'00, vol. 1906 of *LNCS*. Springer, 2000.
3. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.J. Hwang. Symbolic model checking : 10^{20} and beyond. *Information and Computation*, 98:142–170, June 1992.
4. R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: a semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.
5. D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6), November 1995.
6. D. Giannakopoulou, J. Kramer, and S.C. Cheung. Analysing the behaviour of distributed systems using tracta. *Automated Software Engineering*, 6(1):7–35, 1999.
7. S. Graf, B. Steffen, and G. Lüttgen. Compositional minimisation of finite state systems using interface specifications. *Formal Aspects of Computing*, 8(5), 1998.
8. P. Inverardi and S. Uchitel. Proving deadlock freedom in component-based programming. Technical report, Universita' dell'Aquila, Italia, October 1999. http://www.doc.ic.ac.uk/ su2/pubs/techrep99.pdf
9. P. Inverardi, D. Yankelevich, and A. Wolf. Static checking of systems behaviors using derived component assumptions. *ACM TOSEM*, 2000.
10. N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20(2):436–482, March 1998.
11. R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.
12. O.Grumberg and D.E.Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.
13. R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.
14. Clemens Szyperski. *Component Software. Beyond Object Oriented Programming*. Addison Wesley, Harlow, England, 1998.

A Real-Time Execution Semantics for UML Activity Diagrams

Rik Eshuis* and Roel Wieringa

University of Twente, Department of Computer Science
P.O. Box 217, 7500 AE Enschede, The Netherlands
{eshuis,roelw}@cs.utwente.nl

Abstract. We define a formal execution semantics for UML activity diagrams that is appropriate for workflow modelling. Our semantics is aimed at the requirements level by assuming that software state changes do not take time. It is based upon the STATEMATE semantics of statecharts, extended with some transactional properties to deal with data manipulation. Our semantics also deals with real-time and multiple state instances. We first give an informal description of our semantics and then formalise this in terms of transition systems.

1 Introduction

A *workflow* is a set of business activities that are ordered according to a set of procedural rules to deliver a service. A workflow model (or workflow specification) is the definition of a workflow. An instance of a workflow is called a *case*. Examples of cases are an insurance claim handling instance and a production order handling instance. The definition, creation, and management of workflow instances is done by a workflow management system (WFMS), on the basis of workflow models. We represent workflow models by UML activity diagrams [16].

In this paper, we define a formal execution semantics for UML activity diagrams that is suitable for workflow modelling. The goal of the semantics is to support execution of workflow models and analysis of the functional requirements that these models satisfy. Our long term goal is to implement the execution semantics in the TCM case tool [6] and to use model checking tools for the analysis of functional requirements. A secondary goal of the semantics is to facilitate a comparison with other formal modelling techniques for workflows, like Petri nets [1] and statecharts [18]. For example, some people claim that an activity diagram is a Petri net. But in order to sustain this claim, first a formal semantics for activity diagrams must be defined, so that the Petri net semantics and the activity diagram semantics can be compared.

Figure 1 shows an example activity diagram. Ovals represent activity states, rounded rectangles represent wait states, and arrows represent state transitions. Section 3 explains the details of the notation. We use activity diagrams to model a single instance (case) of a workflow. We defer the modelling of multiple cases (case management) to future work.

* Supported by NWO/SION, grant nr. 612-62-02 (DAEMON).

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 76–90, 2001.

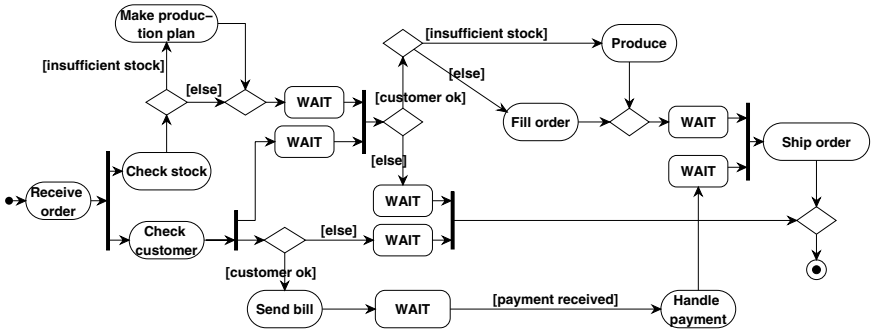


Fig. 1. Example activity diagram

We introduce two semantics. The first semantics supports execution of workflow models. Although this semantics is sufficient for executing workflow models, it is not precise enough for the analysis of functional requirements (model checking), since the behaviour of the environment is not formalised. We therefore define a second semantics, which we will use for model checking, that extends the first one by formalising the combined behaviour of both the system that the activity diagram models and the system's environment.

Our semantics is different from the OMG activity diagram semantics [16], because we map activities into states, whereas the OMG maps them into transitions. The OMG semantics implies that activities are done by the WFS itself, and not by the environment. In our semantics, activities are done by the environment (i.e. actors), not by the WFS itself (see Sect. 2 for our motivation).

The paper is structured as follows. In Sect. 2 we discuss workflow concepts and we give an informal semantics of activity diagrams in terms of the domain of workflow. In Sect. 3 we define the syntax of an activity diagram, and define and discuss constraints on the syntax. In Sect. 4 we define our two formal semantics. In Sect. 5 we briefly discuss other formalisations of activity diagrams. We end with a summary and a discussion of further work. Formulas are written in the Z notation [17].

2 Workflow Domain

Workflow concepts. The following exposition is based on literature (amongst others [1,14]) and several case studies that we did.

Activities are done by *actors*. Actors are people or machines. An *activity* is an uninterruptible amount of work that is performed in a non-zero span of time by an actor. In an activity, *case attributes* are updated. Case attributes are data relevant for the case. They may be present in the form of structured data, or case documents. The *effect* of an activity is constrained declaratively with a pre and post-condition. The pre-condition also functions as guard: as long as it is false, the activity cannot be performed.

The case may be distributed over several actors. Each distributed part of the case has a *local state*. There are three kinds of possible local states.

- In an *activity state* an actor is executing an activity in a part of the case. For every activity there should be at least one activity state, but different activity states can represent execution of the same activity.
- In a *wait state*, the case is waiting for some external event or temporal event.
- In a *queue state*, the case is waiting for an actor to become available to perform the next activity of the case.

Multiple instances of a local state may be active at the same time in the same case. The *global state* of the case is therefore a multiset (rather than a set) of the local states of the distributed parts of the case.

The WFMC [19] specifies four possible ordering relationships between activities: *sequence*, *choice*, *parallelism* and *iteration*. And to facilitate readability and re-use of process definitions, an ordered set of activities can be grouped into one *compound activity*. A compound activity can be used in other process definitions. A non-compound activity is called an *atomic activity*.

System structure (Fig. 2). A workflow system (WFS), which is a WFMS instantiated with one or more workflow models, connects a database and several actors. Since we use an activity diagram to model a single case, we here assume the WFS controls a single case. The WFS routes the case as prescribed by the workflow model of the case. Note that the case attributes are updated during an activity by the actors, not by the WFS. For example, an actor may update a claim form by editing it with a word processor. The state of the case, on the other hand, is updated by the WFS, not by an actor. All attributes of a case are stored in the database. The state of the case is maintained by the WFS itself.

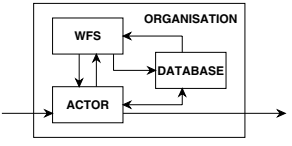


Fig. 2. System structure

Informal semantics. An activity diagram is a requirements specification that says what a WFS should do. We therefore define our semantics of an activity diagram in terms of a WFS.

We view a WFS as a reactive system. A reactive system [12] is a system that runs in parallel with its environment, and reacts to the occurrences of certain events in its environment by creating certain desirable effects in that environment. There are three kinds of events:

- An *external event* is an instantaneous, discrete change of some condition in the environment. This change can be referred to by giving a name to the change itself or to the condition that changes:
  - A *named external event* is an event that is given an unique name.
  - A *value change event* is an event that represents change of one or more variables.
- A *temporal event* is a moment in time, to which the system is expected to respond, i.e. some deadline has been reached.

The behaviour of a reactive system is modelled as a set of runs. A *run* is a sequence of system states and system reactions. System reactions are caused by the occurrence of events.

We make the following two assumptions. First, the goal of our semantics is to support the specification of functional requirements. Functional requirements should be specified independently of the implementation platform, and we therefore make the *perfect technology* assumption: the implementation consists of infinitely many resources that are infinitely fast [15]. Perfect technology implies that the WFS responds infinitely fast to events. This means that the transitions between the local states of a case take no time. But since actors are not assumed to be perfect, they do take time to perform their activities. Second, the WFS responds as soon as it receives events from the environment. This is called the *clock-asynchronous semantics* [11]. These two assumptions together imply that the WFS responds at the same time events occur. This is called the *perfect synchrony hypothesis* [3].

For an implementation of the WFS, these two assumptions translate into the following requirement:

> the WFS must be fast enough in its reaction to the current events to be ready before the next events occur

For runs, these two assumptions imply that time elapses in states only, not in reactions, since reactions are instantaneous, and that there elapses no time between the occurrence of events and the subsequent reaction of the system.

The state of a run of the WFS consists of the following components:

- the global state of the case, including an indication of which activities are currently being performed,
- the current set of input events,
- the current value of the case attributes of the case,
- the current value of the running timers. These are necessary to generate time-outs.

The global state of the case is the multiset of the local states of the individual parallel branches that are active. We call such a global state a *configuration*. Each parallel branch has three kinds of possible states, namely activity, wait and queue states. For the remainder of this paper, we do not consider queue states anymore; we simply assume that there enough actors available for every activity. This is not a serious restriction, because a queue state can be modelled as a wait state, where the event that has to be waited for is that the actor becomes available.

During a reaction, the state of the case is updated (i.e. the case is routed), some timers may be reset, and the set of input events is reset, but the case attributes are not changed. Case attributes are updated by actors during an activity state. During a reaction the current time and the timers do not increase, because a reaction is instantaneous.

We adopt the STATEMATE [11] semantics of a reaction and extend it below with some transactional properties. Before the events occur, the system is in a stable state. When the events occur, the system state has become unstable. To reach a stable state again, the system *reacts* by taking a *step* and entering a new

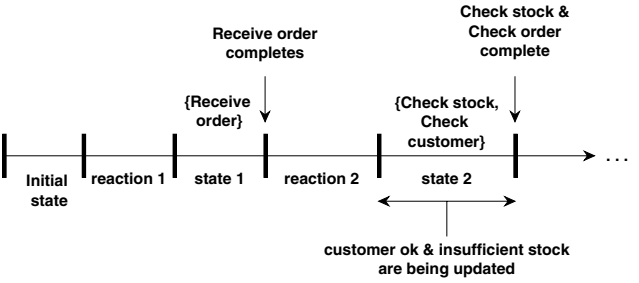


Fig. 3. Run of our example. In each state, the set of activities currently executing is shown

state. If the new state is unstable, again a step is taken, otherwise the system stops taking steps. This sequence of taking a step and entering a new state and testing whether the new state is stable, is repeated until a stable state is reached. Thus, the system reaction is a sequence of steps, called a *superstep* [11]. In Sect. 4 we define steps and supersteps. For the initial state, we do not have to wait for a change in the environment since the initial state is unstable.

Next, we adopt the following assumptions from STATEMATE [11]:

- More than one event can be input to the system at the same time.
- The input is a set of events, rather than a queue (the latter assumption is adopted by the OMG semantics [16], but is more appropriate for a software implementation-level semantics).
- If the system reacts, it reacts to all events in the input set.
- Events live for the duration of one step only (not a superstep!).

Figure 3 shows part of a run of our example. In each state of the run the set of currently executing activities is shown. Both customer ok and insufficient stock are case attributes. See Sect. 4 for details on how a run is constructed.

Specifying activities. An activity has a pre and post-condition. The pre-condition specifies when the activity is allowed to start. The post-condition specifies constraints on the result of the activity. Both pre and post-conditions only refer to case attributes. A case attribute can either be *observed* in an activity by an actor, i.e., used but not changed, or *updated* in an activity by an actor. This may result in an ill-defined case attribute, if the attributes is accessed in two or more concurrently executing activities, and in addition one of the activities updates it. Then the activities *interfere* with each other. Non-interference checks can prevent this. We define an non-interference check on activities, since we view an activity as atomic. An activity may consist of many database transactions (cf. Fig. 2), so non-interference of data manipulation in an activity cannot be handled by a single transaction of the database. Instead, we assume that non-interference checks are done by the WFS (for example by means of a transaction processing monitor that is part of the WFS).

3 Syntax of Activity Diagrams

A UML activity diagram is a graph, consisting of state nodes and directed edges between these state nodes. There are two kinds of state nodes: ordinary state nodes and pseudo state nodes. We follow the UML in considering pseudo state nodes as syntactic sugar to denote hyperedges. Thus, the underlying syntactic structure is a hypergraph, rather than a graph. In order to be unambiguous, we call the underlying hypergraph an *activity machine*.

UML constructs used (Fig. 4). We use an action state node to denote an activity state, a wait state node to denote a wait state, and a subactivity state node to denote a compound activity state. We assume that for every compound activity state node, there is an activity diagram that specifies the behaviour of the compound activity. The transitive closure of this hierarchy relation between activity diagrams must be acyclic. Besides these state constructs, we use pseudo state nodes to indicate xor split and merge (decision state node, merge state node), parallelism (fork state node, join state node), begin (initial state node) and end (final state node). Combining fork and merge, we can specify workflow models and patterns in which multiple instances of the same state node are active at the same time [2]. State nodes (including pseudo state nodes) are linked by directed, labelled edges (expressing sequence). Each label has the form $e[g]$ where e is an event expression and g a guard expression. Empty event NULL and guard [true] are not shown on an edge. Special event label after(*texp*) denotes a relative temporal event, where *texp* is an integer expression. See our report [8] for other temporal constructs (e.g. when).

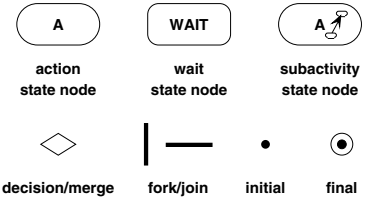


Fig. 4. UML activity diagram constructs used

UML constructs removed

- We do not label edges with action expressions. Actions are performed by actors in activities, not by the WFS in the transitions of a workflow.
- We do not consider synch (synchronisation) states. We have never seen an example of a synch state in our own or other people's case studies.
- We do not consider deferred events (deferring an event means postponing the response to an event). Deferral of event e can be simulated by using the guard [e occurred].
- Swimlanes allocate activities to packages, actors, or organisational units. We disregard swimlanes, since these do not impact the execution semantics. We plan to consider allocation of activities to actors at a later stage.
- The UML includes object flow states, that denote data states. They are connected to other state nodes by object flows (dashed edges). There are several ambiguities concerning object flow states, the most important one being that the meaning of parallel object flow states is not defined. Besides, only one vendor of workflow management systems supports object flow states [14]. For

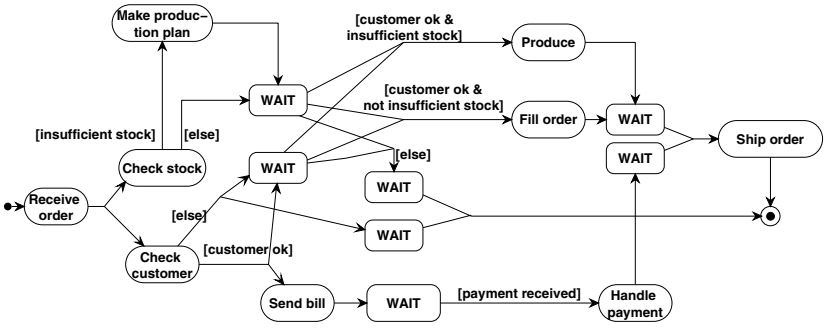


Fig. 5. Activity machine of our running example

the moment, we decide to omit object flow states (and thus object flows) from our syntax. Instead, we represent the case attributes by the local variables of the activity diagram and assume these attributes are stored in a database.
– Dynamic concurrency (i.e. dynamic instantiation of multiple instances of the same action or subactivity state node) we treat in our full report [8].

Syntax of activity machines. An activity machine is a rooted directed hypergraph. Figure 5 shows the activity machine corresponding to Fig. 1. We assume given a set *Activities* of activities. An *activity machine* is a quintuple $(Nodes, Edges, Events, Guards, LVar)$ where:

– $Nodes = AS \cup WS \cup \{initial, final\}$ is the set of state nodes,
– $Edges \subseteq \mathbb{P}\, Nodes \times Events \times Guards \times \mathbb{P}\, Nodes$ is the transition relation between the state nodes of the activity diagram,
– *Events* is the set of external event expressions,
– *Guards* is the set of guard expressions,
– *LVar* is the set of local variables. The local variables represent the case attributes. We assume that every variable in a guard expression is a local variable.

State nodes *initial* and *final* denote the initial and final state node, respectively. Besides these special state nodes, an activity machine has action state nodes AS and wait state nodes WS. Every action state node has an associated activity it controls, denoted by the surjective function $control : AS \twoheadrightarrow Activities$. The execution of the activities falls outside the scope of the activity machine, since it is done by actors. We use the convention that in the activity diagram, an action state node a is labelled with the activity $control(a)$ it controls. Note that different action state nodes may bear the same label, since they may control the same activity. Wait state nodes are labelled WAIT. Edges are labelled with events and guard expressions out of sets *Events* and *Guards* respectively. A special element of *Events* is the empty event NULL, which is always part of the input. Given $e = (N, ev, g, N') \in Edges$, we define $source(e) \stackrel{\text{df}}{=} N$, $event(e) \stackrel{\text{df}}{=} ev$, $guard(e) \stackrel{\text{df}}{=} g$, and $target(e) \stackrel{\text{df}}{=} N'$.

We require that the initial state node only occurs in the source of an edge. Moreover, if it is source of an edge, it is the only source of that edge. Similarly, the final state node may only occur in the target of an edge. Moreover, if it is target of an edge, it is the only target of that edge. Next, the initial state must be unstable. Therefore, the edges leaving the initial state node must be labelled with the empty event NULL and the disjunction of the edges' guard expressions must be a tautology.

Let set $BE(LVar)$ denote the set of boolean expressions on set $LVar$. Next we define the infinite set of all possible $Timers \stackrel{\text{df}}{=} \{t(e)(n) \mid e \in Edges \wedge n \in \mathbb{N}\}$. Roughly, a timer $t(e)(n)$ is reset to zero every time the source states of e are entered. We assume a set $BE(Timers)$ of basic clock constraints on timers. Every basic clock constraint $\phi \in BE(Timers)$ has the form $c = texp$ where $c \in Timers$ and $texp \in \mathbb{N}$.

Set $Guards$ is constructed as the union of $BE(LVar)$ and $BE(Timers)$ and the set of expressions that is obtained by conjoining ($\wedge$) elements of the sets $BE(LVar)$ and $BE(Timers)$. We require that if $t(e)(n) = texp$ is part of the guard expression of edge e', then $e = e'$.

Mapping an activity diagram to an activity machine. In the mapping, first the subactivity state nodes are eliminated by substituting for every subactivity state node its corresponding activity diagram. Then the pseudo state nodes are removed and replaced by hyperedges. Finally, every hyperedge e with label after($texp$) is replaced by infinitely many hyperedges $e(n)$ where $n \in \mathbb{N}$, each edge labelled with clock constraint $t(e)(n) = texp$. (This construction is needed, because a finite but unbounded number of instances of e may be taken simultaneously at the same time.) Our full report [8] gives more details.

Specifying data manipulation in activities. The local variables of the activity diagram are possibly updated in activities (since local variables represent case attributes). In every activity $a \in Activities$ that is controlled by an activity diagram, some local variables may be *observed* or *updated*. We denote the observed variables by $Obs(a) \subseteq LVar$, and the updated variables by $Upd(a) \subseteq LVar$. We require these two sets to be disjoint for each activity.

Two activities *interfere* with each other, if one of them observes or updates a local variable that the other one is updating. (This definition is similar to the definition of conflict equivalence in database theory [7].) Note that in particular every activity only non-interferes with itself iff it only observes variables.

$$\begin{aligned} A \perp B \Leftrightarrow\ & (Obs(A) \cup Upd(A)) \cap Upd(B) \not= \varnothing \\ \vee\ & (Obs(B) \cup Upd(B)) \cap Upd(A) \not= \varnothing \end{aligned}$$

Furthermore, we define for every activity a a pre and post-condition, pre(a) and post(a). The precondition only refers to variables in $Obs(a) \cup Upd(a)$. The post-condition only refers to variables in $Upd(a)$, since the observed variables are not changed.

We assume a typed data domain $\mathcal{D}$. Let $\sigma : LVar \to \mathcal{D}$ be a total, type-preserving function assigning to every local variable a value. We call such a function a *valuation*. Let $\Sigma(LVar)$ denote the set of all valuations on $LVar$. A

partial valuation is a valuation that is a partial function. The set of all partial valuations we denote by $\Sigma_{\mathrm{p}}(LVar)$. A pre or post-condition c always is evaluated w.r.t. some (partial) valuation σ. We write $\sigma \models c$ if c is true in σ (with for every variable v its value $\sigma(v)$ substituted). We do not define the syntax of c: this depends on the data types used. Since we have defined no formal syntax for pre and post-conditions, we do not provide a formal semantics for the satisfaction relation $\models$. In our semantics we simply assume that a formal syntax and semantics for pre and post-conditions has been chosen.

Function $\mathit{effect} : \Sigma_{\mathrm{p}}(LVar) \times AS \nrightarrow \mathbb{P}\,\Sigma(LVar) - \{\varnothing\}$ is a partial function constraining the possible effects of each activity on the case attributes. For a given activity a and partial valuation $\sigma \in \Sigma(Obs(a) \cup Upd(a))$, the set of possible valuations is $\mathit{effect}(\sigma, a) = \{\sigma' \mid \sigma' \models post(a) \wedge \forall\, v \in Obs(a) \bullet \sigma(v) = \sigma'(v)\}$.

Since we allow multiple instances of an activity to be executing, we work with multisets of activities. Given a multiset of activities A that do not interfere, if $\mathit{effect}(\sigma, a)$ is a possible effect of activity a, the combined effect of activities in A is $\uplus_{a \sqsubseteq A} \mathit{effect}(\sigma, a)$, where $\sqsubseteq$ is the membership predicate for multisets and $\uplus$ is union on multisets. Due to the non-interference constraint, the only overlap can be in observed variables and these remain unchanged. We denote the set of possible combined effects with $\mathit{effect}(\sigma, A)$.

Finally, we lift all functions with domain *Activities* to the domain of action state nodes AS by means of function $\mathit{control} : AS \twoheadrightarrow \mathit{Activities}$, defined above. For example, for $a \in AS$, $\mathit{effect}(\sigma, a) \stackrel{\mathrm{df}}{=} \mathit{effect}(\sigma, \mathit{control}(a))$.

Data-related syntactic constraints (Fig. 6). First, every action state node must be followed by a subsequent wait state node. Our semantics will ensure that this wait state node is left iff all variables that have to be tested are not being updated anymore. This prevents a case from being illegally routed. Second, we do not consider pre-conditions in our semantics, since for every activity A that is followed by an activity B if guard g is true, we have that [g] ⇒ post(A) and post(A) ⇒ pre(B). We conclude [g] ⇒ pre(B). So we drop the precondition of an activity from our semantics, since it is already implied by the preceding guard.

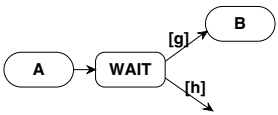


Fig. 6. Modelling pre and post-conditions

4 Execution Semantics of Activity Machines

We give two formal semantics for activity machines that are based upon the informal semantics of Sect. 2. The first semantics defines how a step is computed. This semantics can be used to execute an activity machine, but is not a complete definition of a run. The second semantics, useful for model checking, extends the first one by defining a transition system, whose execution paths are runs. The transition system we use is a clocked Kripke structure. Both semantics are an adaptation of the semantics we defined earlier for UML statecharts [9].

4.1 Computing a Step

States. At each point in time, a system state consists of the current configuration C, the current input I, the current value of every local variable $v \in LVar$, and the valuations of the set $On \subseteq Timers$ of running timers. So the system state is a valuation $\sigma : \{C, I\} \cup LVar \cup On \to \mathcal{D}$.

The configuration is a multiset of state nodes $Nodes \to \mathbb{N}$ of the activity machine. A configuration is non-interfering, written $non\text{-}interfering(C)$, iff it does not contain interfering action state nodes:

$$non\text{-}interfering(C) \stackrel{\text{df}}{\Leftrightarrow} \forall\, a, a' \sqsubseteq C \bullet \neg\, (a \perp a')$$

where as before $\sqsubseteq$ denotes the membership predicate for multisets. In the sequel, the only configurations we allow are non-interfering ones.

Input. We define input I to be a tuple $(Ev, \sigma_{\text{p}}^{LV}, T, \sigma_{\text{p}}^{\text{t}}) \in Events \times \Sigma_{\text{p}}(LVar) \times (AS \to \mathbb{N}) \times \Sigma_{\text{p}}(Timers)$, where as before $\Sigma(S)$ denotes the set of all valuations on set S. Set Ev is the set of external events. We require that empty event NULL is always input: $\{NULL\} \subseteq Ev$. Partial valuation σ_{p}^{LV} represents the set of value change events. The partial valuation assigns to every local variable that is changed its new value. A special value change event occurs when an action state node has terminated because its corresponding activity has completed. This event is modelled by T, which denotes the multiset of terminated action state nodes. We require that only action state nodes in the configuration can terminate: $T \sqsubseteq (AS \lhd C)$, where $\sqsubseteq$ denotes the sub-multiset relation and $X \lhd Y$ denotes restriction of relation Y to the domain set X. Finally, partial valuation $\sigma_{\text{p}}^{\text{t}}$ represents the set of temporal events. A temporal event occurs because some running timer has reached a certain desired value. We therefore require that every timer in the domain of $\sigma_{\text{p}}^{\text{t}}$ is running: $(On \lhd \sigma_{\text{p}}^{\text{t}}) \subseteq \sigma_{\text{p}}^{\text{t}}$.

Steps. A step is a maximal, consistent sub-multiset of the multiset of enabled edges. In addition, the new configuration must be non-interfering. Our definitions extend and generalise both the STATEMATE semantics [11] and our semantics for UML statecharts [9] from sets of states (edges) to multisets of states (edges).

Before we define the multiset of enabled edges $En(C, Ev, T)$, we observe that an edge leaving an action state node a is only enabled if a has terminated, since otherwise the corresponding activity would not be atomic. Therefore, in the definition we do not consider the current configuration C, but instead the multiset C' of non-action state nodes in the current configuration joined with the multiset T of terminated action state nodes. An edge e is n times enabled in the current state σ of the system iff $source(e)$ is contained in C', one of the input events is $event(e)$, the guard can be safely evaluated (denoted by predicate $eval$) and moreover evaluates to true (denoted $\models$) given the current values of all variables, and n is the minimum number of instances of the source state nodes that can be left. Predicate $eval$ states that a guard g can be safely evaluated iff it does not refer to variables that are being updated in the current valuation σ in some activities. We do not refer to σ_{p}^{LV} and $\sigma_{\text{p}}^{\text{t}}$ because these are contained in σ. In formulas:

$$\begin{aligned} En(C, Ev, T) \stackrel{\text{df}}{=} \{ e \shortmid\!\!\rightarrow n \mid\ & ms(source(e)) \sqsubseteq C' \wedge event(e) \in Ev \\ & \wedge\ eval(\sigma, guard(e)) \wedge \sigma \models guard(e) \\ & \wedge\ n = min(\{ C' \sharp s \mid s \in source(e) \}) \ \ \} \\ & \text{where } C' = (((Nodes - AS) \lhd C) \uplus T) \end{aligned}$$

$$eval(\sigma, g) \quad \stackrel{\text{df}}{\Leftrightarrow} \forall\, a \sqsubseteq AS \lhd (C \uplus T) \bullet var(g) \cap Upd(a) = \varnothing$$

where $\uplus$ denotes multiset union, $M \sharp x$ is the number of times x appears in multiset M, $ms(S) \stackrel{\text{df}}{=} \{ s \shortmid\!\!\rightarrow 1 \mid s \in S \}$ (this coerces a set into a multiset), $\uplus$ is difference on multisets, and $var(g)$ denote the set of variables that g tests.

Given configuration C, a multiset of edges E is defined to be consistent, written $consistent(C, E)$, iff all edges can be taken at the same time, i.e. taking one does not disable another one:

$$consistent(C, E) \stackrel{\text{df}}{\Leftrightarrow} (\uplus_{e \sqsubseteq E} ms(source(e))) \sqsubseteq C$$

The function *nextconfig* returns the next configuration, given a configuration C and a consistent multiset of edges E:

$$nextconfig(C, E) \stackrel{\text{df}}{=} C \uplus \biguplus_{e \sqsubseteq E} ms(source(e)) \ \uplus \ \biguplus_{e \sqsubseteq E} ms(target(e))$$

Below, we require that taking a step leads to a non-interfering new configuration.

A multiset of edges E is defined to be maximal iff for every enabled edge e that is added to E, multiset $E \uplus [\![e]\!]$ is inconsistent or the resulting configuration is interfering. Notation $[\![e]\!]$ denotes a bag that contains e only.

$$\begin{aligned} maximal(C, Ev, T) \stackrel{\text{df}}{\Leftrightarrow} \forall\, e \sqsubseteq En(C, Ev, T) \mid e \not\sqsubseteq E \bullet \neg\ & consistent(C, E \uplus [\![e]\!]) \\ & \vee \neg\ non\text{-}interfering(nextconfig(C, E)) \end{aligned}$$

Finally, predicate *isStep* defines a multiset of edges E to be a step iff every edge in E is enabled, E is maximal and consistent, and the next configuration is noninterfering. In our full report [8], this definition is written out as a step algorithm. In the next subsection, we will use the predicate *isStep* again.

$$\begin{aligned} isStep(E) \stackrel{\text{df}}{\Leftrightarrow} & E \sqsubseteq En(C, Ev, T) \wedge consistent(C, E) \\ & \wedge\ maximal(C, Ev, T) \wedge non\text{-}interfering(nextconfig(C, E)) \end{aligned}$$

satisfies *isStep*. So, the system can be nondeterministic.

4.2 Transition System Semantics of Activity Machines

A Clocked Kripke Structure (CKS) is a quadruple $(Var, \rightarrow, ci, \sigma_0)$ where:

- $Var = \{C, Ev, T\} \cup LVar \cup On$ is the set of variables,
- $\rightarrow \subseteq \Sigma(Var) \times \Sigma(Var)$ is the transition relation,
- ci is the clock invariant, a constraint that must hold in every valuation,
- $\sigma_0 \in \Sigma(Var)$ is the initial valuation.

We have omitted from *Var* the *I* components σ_{p}^{LV} and $\sigma_{\mathrm{p}}^{\mathrm{t}}$ since these are already modelled by *LVar* and *On* respectively.

Given an activity machine, its CKS is constructed as follows. First, we specify the clock invariant. For every basic clock constraint $t(e)(n) = texp$, we specify a constraint ϕ of the form $t(e)(n) \in On \Rightarrow t(e)(n) \leq texp$. Clock invariant ci is the conjunction of all constraints ϕ. We evaluate a clock invariant ci in a valuation σ, and write $\sigma \models ci$ if the clock invariant is true.

The transition relation $\rightarrow$ is specified as the union of three other transition relations. Not every sequence of transitions out of this union satisfies the clock-asynchronous semantics. Every valid sequence must start with a superstep (the initial step) followed by a sequence of cycles. The initial superstep is taken because the initial state is by definition unstable. A cycle is one or more time steps, followed by an event step, followed by a superstep. Note that $\rightarrow_{cycle}$ is not part of $\rightarrow$.

$$\rightarrow_{cycle} \stackrel{\mathrm{df}}{=} \rightarrow_{timestep}^{+} \fatsemi \rightarrow_{event} \fatsemi \rightarrow_{superstep}$$

Relation $\rightarrow_{timestep}$ represents the elapsing of time by updating timers with a delay Δ such that the clock invariant is not violated. In the following, $\&_{s \in S}$ denotes a concurrent update done for all elements of set S.

$$\begin{aligned}\sigma \rightarrow_{timestep} \sigma' \stackrel{\mathrm{df}}{\Leftrightarrow} & \exists\, \Delta \in \mathbb{R} \mid \Delta > 0 \bullet \sigma' = \sigma[\&_{c \in On}\, c / \sigma(c) + \Delta] \\ & \text{such that } \forall\, \delta \in [0, \Delta] \bullet \sigma[\&_{c \in On}\, c / \sigma(c) + \delta] \models ci\end{aligned}$$

Relation $\rightarrow_{event}$ defines that events occur between σ and σ' iff timers did not change, the configuration did not change, and the local variables that are being updated in activities are not changed, and:

- either there are named external events in the input in state σ';
- or there is a nonempty set L of local variables that have no interference with the currently executing activities and whose value changed;
- or some action state nodes have terminated, and there is a partial valuation $\sigma'_{\mathrm{p}} \subset \sigma'$ that conforms to the effect constraints of the currently terminating activities;
- or it is not possible to do any more time steps (i.e. a deadline is reached).

$$\begin{aligned}\sigma \rightarrow_{event} \sigma' \stackrel{\mathrm{df}}{\Leftrightarrow} & (\forall\, c \in On \bullet \sigma(c) = \sigma'(c)) \wedge \sigma(C) = \sigma'(C) \\ & \wedge\ \forall\, v \in LVar;\ \forall\, a \in AS \mid a \sqsubseteq \sigma(C) \uplus \sigma'(T) \bullet \\ & \qquad v \in Obs(a) \cup Upd(a) \Rightarrow \sigma(v) = \sigma(v') \\ & \wedge\ (\quad \sigma'(Ev) \subseteq Events \wedge \{NULL\} \subset \sigma'(Ev) \\ & \quad \vee\ \exists\, L \subseteq LVar \mid L \not= \varnothing \bullet \\ & \qquad (\forall\, a \in AS \mid a \sqsubseteq \sigma(C) \bullet L \cap (Obs(a) \cup Upd(a)) = \varnothing) \\ & \quad \vee\ [\![]\!] \not= \sigma'(T) \sqsubseteq (AS \lhd \sigma(C)) \wedge \exists\, \sigma'_{\mathrm{p}} \in effect(\sigma, \sigma'(T)) \bullet \sigma'_{\mathrm{p}} \subset \sigma' \\ & \quad \vee\ \nexists \sigma'' \bullet \sigma \rightarrow_{timestep} \sigma'' \quad)\end{aligned}$$

Finally, a superstep is a sequence of steps, such that intermediate states are unstable and the final state of the sequence is stable. The notation $f \oplus g$

means that function g overrides function f on the domain of f. Note that the intermediary states (the semicolon in the composition of the relations) are not part of the CKS.

$$
\begin{aligned}
\rightarrow_{superstep} &\stackrel{\text{df}}{=} (\rightarrow_{unstable} \fatsemi \rightarrow_{step} \fatsemi \rightarrow_{superstep}) \cup \rightarrow_{stable} \qquad (1)\\
\sigma \rightarrow_{step} \sigma' &\stackrel{\text{df}}{\Leftrightarrow} \exists\, E \mid isStep(E) \bullet\\
&\qquad \exists\, S_1 \subseteq Timers \mid OffTimers(\sigma(C), E, \sigma(On), S_1);\\
&\qquad \exists\, S_2 \subseteq Timers \mid NewTimers(\sigma(C), E, \sigma(On), S_2) \bullet\\
&\qquad\quad \sigma' = \sigma[C/nextconfig(\sigma(C), E), Ev/\varnothing, T/[\![]\!],\\
&\qquad\qquad \&_{s \in S_2} s/0, On/\sigma(On) - S_1 \cup S_2]\\
\sigma \rightarrow_{unstable} \sigma' &\stackrel{\text{df}}{\Leftrightarrow} \sigma = \sigma' \wedge En(\sigma(C), \sigma(Ev), \sigma(T)) \not= \varnothing\\
\sigma \rightarrow_{stable} \sigma' &\stackrel{\text{df}}{\Leftrightarrow} \sigma = \sigma' \wedge En(\sigma(C), \sigma(Ev), \sigma(T)) = \varnothing
\end{aligned}
$$

Line by line, the $\rightarrow_{step}$ definition says that a step is done between σ and σ' iff:

- there is a step E (using the predicate *isStep* defined in Sect. 4.1);
- there is a set S_1 of timers that can be turned off (denoted by predicate *OffTimers*;
- there is a set S_2 of timers that can be turned on (denoted by predicate *NewTimers*;
- σ is then updated into σ' by computing the next configuration when step E is performed (using the function *nextconfig* defined in Sect. 4.1), and resetting the input, the multiset of terminated action state nodes, and all the new timers in S_2, and finally updating *On*.

Predicates $\rightarrow_{unstable}$ and $\rightarrow_{stable}$ test whether there are enabled edges. We compute a superstep by taking a least fixpoint of (1). This may not exist; in which case the superstep does not terminate. Or it may not be unique, in which case there is more than one possible superstep.

We now proceed to define predicates *OffTimers* and *NewTimers*. First we define the notion of relevant hyperedges. Given configuration C, the multiset of relevant hyperedges $rel(C)$ contains each edge whose source is contained in C.

$$rel(C) = \{e \mapsto n \mid source(e) \subseteq C \wedge n = min(\{C \sharp s \mid s \in source(e)\})\}$$

For every relevant edge with a clock constraint a timer is running. This timer was started when the edge became relevant. It will be stopped when the edge will become irrelevant. Assume given a configuration C, a step E, a set of running timers *On*, and a set S of timers. Predicate *OffTimers* is true iff all timers in S are running, but can now be turned off, because their corresponding edges are relevant for C, but are no longer relevant if E is taken. Predicate *NewTimers* is true iff all timers in S are off, but can now be turned on, because their corresponding edges are irrelevant for C, but do become relevant if E is taken.

$$
\begin{aligned}
OffTimers(C, E, On, S) \;&\Leftrightarrow S \subseteq On \cap \{t(e)(n) \mid e \sqsubseteq R \wedge n \in \mathbb{N}\}\\
&\quad \wedge\; \forall\, e \sqsubseteq R \bullet R \sharp e = \#(S \cap \{t(e)(n) \mid n \in \mathbb{N}\})\\
&\qquad \text{where } R = rel(C) - rel(nextconfig(C, E))
\end{aligned}
$$

$$NewTimers(C, E, On, S) \Leftrightarrow S \subseteq (Timers - On) \cap \{t(e)(n) \mid e \sqsubseteq R \wedge n \in \mathbb{N}\}$$
$$\wedge\ \forall\, e \sqsubseteq R \bullet R \sharp e = \#(S \cap \{t(e)(n) \mid n \in \mathbb{N}\})$$
$$\text{where } R = rel(nextconfig(C, E)) - rel(C)$$

In the initial valuation σ_0, the configuration only contains one copy of *initial*, the only input event is NULL, and *On* is empty.

5 Related Work

The OMG [16] gives a semantics to an activity diagram by translating it into a UML statechart. Both the translation and the semantics of UML statecharts are not formally defined. Moreover, the translation is inappropriate, since activity diagrams are more expressive than statecharts. The OMG semantics (and other semantics too [4,5]) maps action state nodes to transitions. This means that updates to case attributes are made by the WFS itself, and not by the actors. Our semantics maps action state nodes to states (valuations), which means that activities are performed by actors, not by the WFS.

Gehrke *et al.* [10] give a semantics by translating an activity diagram into a Petri net. Their semantics does not deal with data or time as we do. In addition, Petri net semantics do not model the environment, whereas our semantics does model the environment. We provide a more detailed comparison with all these other formalisations in our full report [8].

6 Conclusions and Future Work

We have defined a formal real-time requirements-level execution semantics for UML activity diagrams that manipulate data, for the application domain of workflow modelling. The semantics is based on the STATEMATE statechart semantics, extended with transactional properties. We defined both an execution and a transition system semantics. Our semantics is different from other proposed semantics, both for activity diagrams and for workflow models. It is motivated by analysis of the workflow literature and by case studies.

We have done initial experiments with model checking simple statecharts using the model checker Kronos [13]. Future work includes extending this to model checking activity machines. Next, we plan a detailed comparison with Petri net semantics.

References

1. W.M.P. van der Aalst. The application of Petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
2. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Advanced workflow patterns. In O. Etzion and P. Scheuermann, editors, *Proc. CoopIS 2000*, LNCS 1901. Springer, 2000. Workflow pattern home page: `http://www.mincom.com/mtrspirt/workflow`.

3. G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Sci. of Comp. Prog.*, 19(2):87–152, 1992.
4. C. Bolton and J. Davies. Activity graphs and processes. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Proc. Integrated Formal Methods 2000*, LNCS 1945. Springer, 2000.
5. E. Börger, A. Cavarra, and E. Riccobene. An ASM Semantics for UML Activity Diagrams. In T. Rus, editor, *Proc. Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000*, LNCS 1826. Springer, 2000.
6. F. Dehne, R. Wieringa, and H. van de Zandschulp. Toolkit for conceptual modeling (TCM) — user's guide and reference. Technical report, University of Twente, 2000. `http://www.cs.utwente.nl/~tcm`.
7. R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Benjamins/Cummings, Redwood City, 1989.
8. R. Eshuis and R. Wieringa. A formal semantics for UML activity diagrams, 2001. Available at `http://www.cs.utwente.nl/~eshuis/adsem.ps`.
9. R. Eshuis and R. Wieringa. Requirements-level semantics for UML statecharts. In S.F. Smith and C.L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems IV*, pages 121–140. Kluwer Academic Publishers, 2000.
10. T. Gehrke, U. Goltz, and H. Wehrheim. The dynamic models of UML: Towards a semantics and its application in the development process. Hildesheimer Informatik-Bericht 11/98, 1998.
11. D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.
12. D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, NATO/ASI 13. Springer, 1985.
13. D. N. Jansen and R. J. Wieringa. Extending CTL with actions and real-time. In *Proc. International Conference on Temporal Logic 2000*, 2000.
14. F. Leymann and D. Roller. *Production Workflow — Concepts and Techniques*. Prentice Hall, 2000.
15. S. McMenamin and J. Palmer. *Essential Systems Analysis*. Yourdon Press, New York, New York, 1984.
16. OMG. *Unified Modeling Language version 1.3*. OMG, July 1999.
17. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1992.
18. D. Wodtke and G. Weikum. A formal foundation for distributed workflow execution based on state charts. In F. N. Afrati and P. Kolaitis, editors, *6th International Conference on Database Theory (ICDT)*, LNCS 1186. Springer, 1997.
19. Workflow Management Coalition. Workflow management coalition specification — terminology & glossary (WFMC-TC-1011), 1999. `http://www.wfmc.org`.

A CSP View on UML-RT Structure Diagrams

Clemens Fischer, Ernst-Rüdiger Olderog, and Heike Wehrheim

Universität Oldenburg
Fachbereich Informatik
Postfach 2503, D–26111 Oldenburg, Germany
Fax: +49 441 7982965
{fischer,olderog,wehrheim}@informatik.uni-oldenburg.de

Abstract. UML-RT is an extension of UML for modelling embedded reactive and real-time software systems. Its particular focus lies on system descriptions on the *architectural* level, defining the overall system structure. In this paper we propose to use UML-RT structure diagrams together with the formal method CSP-OZ combining CSP and Object-Z. While CSP-OZ is used for specifying the system components themselves (by CSP-OZ classes), UML-RT diagrams provide the architecture description. Thus the usual architecture specification in terms of the CSP operators parallel composition, renaming and hiding is replaced by a *graphical* description. To preserve the formal semantics of CSP-OZ specifications, we develop a translation from UML-RT structure diagrams to CSP. Besides achieving a more easily accessible, graphical architecture modelling for CSP-OZ, we thus also give a semantics to UML-RT structure diagrams.

1 Introduction

Graphical modelling notations are becoming increasingly important in the design of industrial software systems. The Unified Modelling Language (UML [2,13]), being standardised by the Object Management Group OMG, is the most prominent member of a number of graphical modelling notations for object-oriented analysis and design. UML-RT [16] is a UML profile proposed as a modelling language for embedded real-time software systems. Although the name RT refers to real-time, UML-RT's main extension concerns facilities for describing the *architecture* of distributed interconnected systems. UML-RT defines three new constructs for modelling structure: *capsules, ports* and *connectors*, and employs these constructs within UML's collaboration diagrams to obtain an architecture description. The advantage of UML-RT, like UML, is the *graphical* representation of the modelled system. However, it lacks a precise semantics.

A different approach to the specification of software systems is taken when a formal method is used as a modelling language. In contrast to UML, formal methods have a precise semantics, but mostly do not offer graphical means of specification. A joint usage of formal methods and graphical modelling languages could thus benefit from the advantages and overcome the deficiencies of each

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 91–108, 2001.

method. A number of proposals for combining UML with a formal method have already been made (e.g. [3,4,10,14]). This paper makes another contribution in this field, focusing on one particular aspect of system modelling, the *architecture descriptions*.

The formal method we employ is CSP-OZ [5,6], a combination of the process algebra CSP [9,15] and the specification language Object-Z [17,18]. The work presented in this paper can be seen as a first step towards an integration of UML and CSP-OZ. CSP-OZ has several features which makes it a suitable candidate for a formal method supporting UML. To name just two: it is an object-oriented notation (with concepts like classes, instantiation and inheritance), and, like UML, it combines a formalism for describing static aspects (Z) with one for describing the dynamic behaviour (CSP).

CSP-OZ specifications typically consist of three parts: first, some basic definitions of e.g. types are made; second, *classes* are defined, and finally, the *system architecture* (components and their interconnections) is fixed. All ingredients which usually appear in UML class descriptions can be found in CSP-OZ classes: attributes, methods and inherited superclasses are declared, associations can be modelled by using attributes with type of another class. Furthermore, one part of a CSP-OZ class specifies the *dynamic behaviour* of the class, which, in UML, is usually given by a separate diagram, e.g. a state chart. In contrast to UML, CSP-OZ uses the CSP process-algebraic notation for this purpose. The system architecture is given by instantiating the classes into a suitable number of objects and combining them using the CSP operators for parallel composition, hiding and renaming. To clarify this overall structure of the system, often some sort of ad-hoc connection diagram is drawn. But these diagrams only serve as an illustration of the CSP architecture description; neither is the form of the diagrams fixed in any way, nor do they have a formal semantics. Hence they cannot actually *replace* the CSP description.

For the integration of CSP-OZ and UML we start here with this last part of CSP-OZ specifications. Defining the system architecture in the above described sense is exactly the intended purpose of UML-RT structure diagrams. Our proposal in this paper is therefore to replace the textual CSP architecture descriptions by UML-RT structure diagrams. To preserve the precision of a formal method we fix the syntax and semantics of these diagrams. The advantages are twofold: UML-RT provides us with a widely accepted graphical specification technique for defining architectures, and additionally a formal semantics for UML-RT structure diagrams in the setting of distributed communicating systems is achieved. For the other main ingredients of CSP-OZ specifications, the classes, we envisage an integration with UML in the following way: CSP-OZ classes are split into a static part, with an appropriate representation by UML class diagrams, and a dynamic part, with a representation by e.g. an activity diagram or a state chart. However, in this paper we are only concerned with obtaining a graphical description of the architecture specification.

Technically, the use of UML-RT diagrams within CSP-OZ is achieved by the following two issues: a formalisation of the syntax of UML-RT diagrams

with Object-Z and a formalisation of their semantics using the CSP operators for parallel composition, renaming and hiding. The major difficulty in giving a semantics to UML-RT diagrams is the treatment of the *multi-object* notation of UML: a component may communicate with a multi-object (a number of instances of the same class) over a single channel. In this case some kind of *addressing* has to be introduced to achieve communication with a particular instance, i.e. the channel additionally has to carry addresses of receivers. Since the need for introducing an addressing mechanism in communication is already visible in the architecture description, the translation also has to take addressing into account. This is solved by introducing special address parameters for processes and carrying out appropriate renamings on channel names of components.

The case study in Section 4 demonstrates the usefulness of the multi-object notation in a real-life application. In our applications the basic capsules in UML-RT structure diagrams stand for CSP-OZ classes or simply CSP processes. However, in this paper the details of CSP-OZ are not important. Here we only need the architectural operators of CSP that allow us to connect CSP-OZ classes or CSP processes: parallel composition, renaming and hiding.

For UML-RT the results of this paper are a contribution to its formal semantics. We nevertheless believe that this is not the only possible interpretation (in particular since we only use a very simple form of protocols allowing for synchronous communication over one channel); in a different setting a different semantics might be conceivable. For CSP and CSP-OZ the benefit is a precise type of diagram for describing the system architecture, replacing the informal connection diagrams that appear in books on process algebra like [9], viz. UML-RT structure diagrams. Particularly interesting is the use of multi-objects of UML-RT for a concise description of iterated CSP operators.

The paper is organised as follows. The next section gives a short introduction to the specific constructs of UML-RT and formalises the syntax of structure diagrams using Object-Z. Section 3 defines the semantics of these diagrams by a translation to CSP. We illustrate our approach by some smaller examples and, in Section 4, by the case study of an automatic manufacturing system. The conclusion summarises our work and discusses some related approaches.

2 UML-RT

Currently, the UML, as standardised by the OMG, is the most widely used object-oriented modelling language. UML-RT (UML for Real-Time Systems) [16] is an extension of UML designed for describing *architectures* of embedded real-time systems. The emphasis of this extension lies on the modelling of the *structure* of distributed systems, no particular real-time features are added to UML's possibilities. The extension uses standard UML tailoring mechanisms like stereotypes and tagged values.

In the following, we briefly describe UML-RT and give some small examples. UML-RT defines three constructs for modelling the structure of distributed systems:

Capsules describe complex components of systems that may interact with their environment. Capsules may be hierarchically structured, enclosing a number of subcapsules which themselves may contain subcapsules.

Ports: Interaction of capsules with the environment is managed by attaching ports to capsules. They are the only means of interaction with the environment. Ports are often associated with *protocols* that regulate the flow of information passing through a port. Ports can furthermore be either *public* or *private*. Public ports have to be located on the border of a capsule.

Connectors are used to interconnect two or more ports of capsules and thus describe the communication relationships between capsules.

For all three constructs stereotypes are introduced: ≪capsule≫ and ≪port≫ are stereotypes for particular classes, whereas ≪connector≫ is a stereotype used for an association. The ports of a capsule class are listed in a separate compartment after the attribute and operator list compartments. A class diagram can be used to define all capsule classes of a system. The actual architecture of the system is given by a *collaboration diagram*, fixing the components of the system and their interconnections. The actual dynamic behaviour of a capsule that contains no other capsules inside can be given by a state machine. In the context of CSP-OZ we will instead assume to have a CSP description of the dynamic behaviour. The communication paradigm of CSP is *synchronous communication*, thus we also assume this in the following.

In a collaboration diagram capsules are represented by rectangles which may contain other capsules, ports are indicated by small black- or white-filled squares. Figure 1 gives a first example of an UML-RT collaboration diagram.

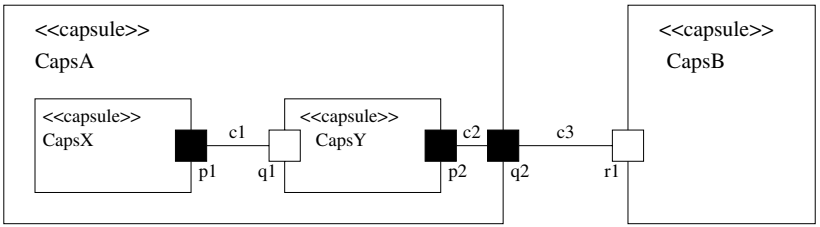


Fig. 1. Collaboration diagram

The collaboration diagram shows a capsule of class *CapsA* consisting of two subcapsules *CapsX* and *CapsY*. These capsules have ports $p1$ and $q1$, which are connected and private within capsule *CapsA*. Capsule *CapsY* furthermore has a port $p2$ connected with the public port $q2$ of capsule *CapsA*. Capsule *CapsA* is connected with capsule *CapsB* via a connector $c3$ between public ports $q2$ and $r1$.

In the case of binary protocols (two participants in the communication), a port and its conjugate part can be depicted by black- and white-filled squares, respectively. In the following we will always use the black-filled port for the sender and the white-filled port for the receiver of messages. This is motivated by our choice of using the CSP communication paradigm.

Another UML notation frequently used in UML-RT collaboration diagrams is the *multi-object* notation (depicted as a stack of rectangles); a number may be used to indicate the actual number of instances. Figure 2 shows the use of the multi-object notation for capsules and ports. Capsule A can communicate with instances of the multi-capsule B via port p. The instance names $\{u, v, w\}$, attached to the multi-capsule via a note, can be used for addressing the desired communication partner. Addressing is needed since there is a single name for the sending port but multiple receivers. The multiplicity of the sending port p indicates this addressing.

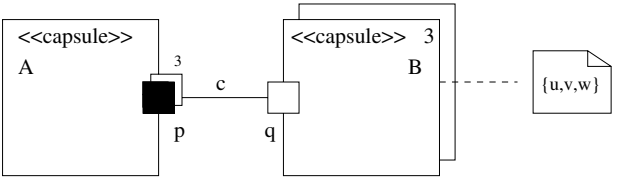


Fig. 2. Multi-object notation for capsules

2.1 Z Formalisation of the Syntax of UML-RT Diagrams

For the translation of UML-RT collaboration diagrams into CSP we need a precise description of their syntax. To this end, we use the formal method Z and its extension Object-Z, which are frequently employed for syntax descriptions of UML diagrams [7,10].

The building blocks of UML-RT collaboration diagrams are ports, connectors, capsules and instances of capsules. For each of these we need a type for their names:

$$[PortName, ConName, CapName, InstName]$$

A *port* is always connected to some capsule. Thus a port has a port name, a capsule name and a *colour*, viz. black or white. The intuition is that input ports are white and output ports are black.

$$Colour ::= black \mid white$$

Furthermore, a port may be a multi-object and thus has a multiplicity specified. This is represented as an Object-Z class

Port

$$name : PortName$$
$$cname : CapName$$
$$colour : Colour$$
$$multi : \mathbb{N}_1$$

$$colour = white \Rightarrow multi = 1$$

with a state consisting of the attributes *cname, colour* and *multi*. We assume that white ports (the receivers) always have multiplicity one. The particular receiver of a message is solely determined by the sender. This restriction is not enforced by UML-RT, but chosen with our particular setting with CSP communication in mind. A port with multiplicity 1 is called *single port* and a port with multiplicity greater than 1 is called a *multi-port*.

A *connector* comprises a name and a finite set of ports. For simplicity we assume that every connector combines exactly two ports. Thus we only consider the case of *binary protocols* or point-to-point communication. The name of a connector is not always given in UML diagrams, but we assume that some unique default name for every connector can be generated. This is represented by an Object-Z class

Connector

$$name : ConName$$
$$ports : \mathbb{F}\, Port$$
$$\#ports = 2$$

where the state has the attributes *name* and *ports*.

In UML-RT system components are represented by capsules. We distinguish between *basic, compound* and *multi-object capsules*. This is formalised using Object-Z by defining a base class *Capsule* which is then extended via inheritance to subclasses *CompCapsule* and *MultiCapsule*. A basic capsule will be translated into a CSP-OZ class or simply a CSP process. It usually has a number of ports to be able to communicate with its environment. A compound capsule has in addition some subcapsules linked by connectors. A multi-object capsule is obtained from a basic or compound capsule by adding a multiplicity.

Each capsule has a name. We assume that capsule names are unique, i.e. there exists an injective function $\mathcal{C}$ from capsules names to capsules (and all their subclasses denoted by the operator $\downarrow$):

$$\mathcal{C} : CapName \rightarrowtail \downarrow Capsule \qquad \text{[Capsule and all its subclasses]}$$
$$\forall\, na : \operatorname{dom} \mathcal{C} \bullet na = \mathcal{C}(na).name \qquad \text{[}\mathcal{C}\text{ respects capsule names]}$$

The function $\mathcal{C}$ is a partial function because not every capsule name has to be used in a given design. We also assume port names and connector names to be unique, which can be similarly fixed by defining partial functions $\mathcal{P}$ for ports and $\mathcal{N}$ for connectors.

A *capsule* itself has a name and a finite set of ports that are considered as *public*, i.e. accessible from the environment. This is represented as an Object-Z base class where the state has the attributes *name* and *ports*:

Capsule

$$
\begin{array}{|l}
\hline
name : CapName \\
ports : \mathbb{F}\, Port \\
\hline
\forall\, p : ports \bullet p.cname = name \\
\hline
\end{array}
$$

The consistency condition requires that all ports of a capsule refer to that capsule.

A *compound capsule* extends a given capsule by an inner structure consisting of a finite set of subcapsules, referenced by names, and a finite set of connectors. We use therefore the inheritance notation of Object-Z[1]:

CompCapsule

`inherit` *Capsule*

$$
\begin{array}{|ll}
\hline
scnames : \mathbb{F}\, CapName & [\text{ names of subcapsules}] \\
conn : \mathbb{F}\, Connector & [\text{inner connectors}] \\
\hline
\forall\, c : conn \bullet c.ports \subseteq ports \cup \bigcup \{sn : scnames \bullet \mathcal{C}(sn).ports\} & [1] \\
\forall\, c_1, c_2 : conn \bullet c_1.ports \cap c_2.ports \not= \varnothing \Rightarrow c_1 = c_2 & [2] \\
\forall\, c : conn \bullet \forall\, p, p' : c.ports \bullet p \not= p' \Rightarrow p.cname \not= p'.cname & [3] \\
\forall\, c : conn \bullet \forall\, p, p' : c.ports \bullet (\ p \not= p' \Rightarrow & \\
\quad (p \in ports \Rightarrow p.multi = p'.multi \wedge p.colour = p'.colour) & [4] \\
\quad \wedge & \\
\quad (p, p' \not\in ports \Rightarrow p.colour \not= p'.colour \wedge & [5] \\
\quad\quad ((p.multi = 1 \wedge p'.multi = \mathcal{C}(p.cname).multi) & \\
\quad\quad\quad \vee & \\
\quad\quad\ (p'.multi = 1 \wedge p.multi = \mathcal{C}(p'.cname).multi))\ )\ ) & \\
\hline
\end{array}
$$

The ports of the subcapsules are treated as *private*, i.e. hidden form the environment. The connectors link these private ports of the subcapsules to each other or to the public ports of the whole capsule. The predicates state consistency conditions for compound capsules:

- connectors can only connect ports given in the capsule (condition 1),
- any two connectors with a common port are identical, i.e. connectors represent point-to-point connections without fan-out (condition 2),
- connectors may only connect ports of different capsules (condition 3).

[1] Inheritance in Object-Z is syntactically expressed by simple inclusion of a class. Semantically inheritance is expressed by signature extension and logical conjunction.

Furthermore the multiplicities of the ports of a connector have to match:

- if one port of the connector is public, the multiplicities and colours of both ports of the connector coincide (condition 4),
- if both ports of the connector are private, the colours of the ports differ, one is a single port and the other one has a multiplicity that coincides with the multiplicity of the subcapsule of the single port (condition 5, see also Figure 2). Note that this covers the case that both ports and hence both their subcapsules are single ones.

A *multi-capsule* inherits all attributes from the base class *Capsule* and furthermore contains another capsule (the one which is multiplied), the multiplicity and the names of the instances (we assume that these names are either given in the diagram, e.g. by attaching a note to the multi-capsule, or can be generated). Using inheritance this can be represented as follows:

MultiCapsule

`inherit` *Capsule*

$$
\begin{array}{|l}
\hline
scname : CapName \\
multi : \mathbb{N}_1 \\
inames : \mathbb{F}\, InstName \\
\hline
\#inames = multi \\
\hline
\end{array}
$$

Starting from ports, connectors and capsules, systems can be built up. A *system* is defined as an outermost capsule.

3 Translating UML-RT Diagrams to CSP

UML-RT structure diagrams give a graphical description of the architecture of systems, their components and interconnections. The Object-Z formalisation so far fixes the valid syntactic structures of these diagrams. For giving a formal semantics of the behavioural aspects of the diagrams, we use another formal method which is particularly suited for the description of distributed communicating systems: the process algebra CSP [9,15].

3.1 A Brief Re-cap of CSP

CSP is a formal method for specifying and reasoning about processes. CSP describes a system as a number of processes, possibly running in parallel and synchronously communicating over named *channels*. Each process is built over some set of communication events, using operators like sequential composition or choice to construct more complex processes. For describing the *architecture* of systems the operators *parallel composition*, *hiding* and *renaming* are used.

Parallel composition, denoted by $\|_A$, is used to set processes in parallel, requiring synchronisation on all communication events in the set A. A communication event consists of a channel name and some values of parameters (e.g. $ch.v_1.v_2$). For instance, the term $C_1 \|_{\{ch.1,ch.2\}} C_2$ describes a parallel composition of components C_1 and C_2 which have to synchronise on the execution of event $ch.1$ (and similar $ch.2$). Often the synchronisation set is simply a set of channels and then stands for synchronisation on all events built over these channel names. The operator $|||$ stands for *interleaving*, i.e. parallel composition with empty synchronisation set. Since interleaving is associative, it can be iterated:

$$\Big|||_{i:I} P_i \qquad \text{where } I \text{ is a finite index set}$$

Alphabetised parallel $P_A \|_B Q$ is another version of parallel composition. If A and B are sets of events, $P_A \|_B Q$ is the combination where P is allowed to communicate in the set A, called the alphabet of P, and Q is allowed in the set B, the alphabet of Q, and both P and Q must agree on events in the intersection $A \cap B$. Also alphabetised parallel can be iterated. For a finite index set I the notation is

$$\Big\|_{i:I}[A_i] \bullet P_i \qquad \text{where } A_i \text{ is the alphabet of } P_i.$$

Hiding, denoted by $\setminus A$, is used to make communication events *internal* to some components. Technically, hiding is achieved by renaming some events into the invisible event τ. Thus they are not available for communication anymore. This corresponds well to the concept of private ports.

Renaming, denoted by $[R]$ where R is a relation between events, is used to rename communication events. The renaming most often only concerns the channel names, not the values of parameters. Therefore, R may also be a relation on channel names. For instance, the term $C[in \mapsto out]$ describes a process C where all communication events of the form $in.x$ are renamed into $out.x$.

3.2 Translation of Examples

In general, every capsule of a diagram stands for an instantiation of a specific CSP process. Ports are modelled by channels, subcapsules within some capsule have to be put into parallel composition with appropriate synchronisation sets guaranteeing the interconnections among capsules as defined by the connectors. If a capsule is basic and contains no further subcapsules, the concrete CSP process remains undefined (since collaboration diagrams do not model the precise behaviour of components, only the structure of the system) and the only part that is used in the architecture description is the name of the capsule. The actual class or process definition behind the capsule has to be specified somewhere else. Given the process names for these basic non-hierarchical capsules, the CSP term for the whole diagram can be inductively constructed.

Before giving a formal definition of the translation of UML-RT diagrams into CSP, we explain the translation informally using the two examples shown in

Figures 1 and 2 of the last section. In Figure 1, there are three basic capsules which do not contain any subcapsules. For these three we assume to have some definition at hand (e.g. as a CSP-OZ class) and just use their names:

$$CapsX, CapsY, CapsB$$

All ports of the three capsules have multiplicity 1. Therefore no addressing is needed here and the process names are not parametrised. For deriving the process term of the compound capsule *CapsA*, we now have to compose processes *CapsX* and *CapsY* in parallel. The choice of the synchronisation set requires some care. If we simply use the names p_1 and q_1 of the ports attached to the connector c_1, no communication is possible at all in the CSP model because *CapsX* communicates only on channel p_1 whereas *CapsY* communicates only on channel q_1. Instead we use the name of the *connector* in the synchronisation set and carry out an appropriate renaming of port names to connector names on the subcapsules by $R_X = \{p_1 \mapsto c_1\}$ and $R_Y = \{q_1 \mapsto c_1, p_2 \mapsto c_2\}$:

$$CapsX[R_X] \;{}_{\{c_1\}}||_{\{c_1,c_2\}}\; CapsY[R_Y]$$

Thus communication between the capsules *CapsX* and *CapsY* is modelled by the CSP paradigm of synchronous (or handshake or rendezvous) communication.

For obtaining the process for the compound capsule *CapsA* two more operations have to be applied: the channel c_2 has to be renamed into q_2 (the name of the public port on the border of *CapsA*) and afterwards the channel c_1 (connector of two private ports) has to be hidden. Summarising, we get the following CSP process for capsule *CapsA*:

$$CapsA = \big(CapsX[R_X] \;{}_{\{c_1\}}||_{\{c_1,c_2\}}\; CapsY[R_Y]\big)[c_2 \mapsto q_2] \setminus \{c_1\}$$

The CSP process describing the complete architecture of the system is then the parallel composition of the two capsules *CapsA* and *CapsB*, again applying renaming, synchronisation and hiding of private ports. *System* can be seen as the compound capsule containing all capsules in the diagram.

$$System = \big(CapsA[q_2 \mapsto c_3] \;{}_{\{c_3\}}\|_{\{c_3\}}\; CapsB[r_1 \mapsto c_3]\big) \setminus \{c_3\}$$

The example in Figure 2 requires a careful treatment of addressing. Port p with multiplicity 3 indicates the possibility of capsule A to choose between different receivers of messages sent over p. This is modelled by parameterising the process name of A with a formal parameter Adr_p standing for the set of possible receivers: $A(Adr_p)$. For example, suppose the CSP process of capsule A is

$$A(Adr_p) = Produce(e);\ (\ |||_{out:Adr_p}\ p.out!e);\ A(Adr_p)$$

$A(Adr_p)$ repeatedly produces an element e which is then output in parallel to all addresses *out* of the set Adr_p via the multi-port p. Instantiating the formal parameter Adr_p with the address set $\{u, v, w\}$ yields

$$A(\{u, v, w\}) = Produce(e);\ (p.u!e \;|||\; p.v!e \;|||\; p.w!e);\ A(\{u, v, w\}).$$

Basic capsule B is treated as before. For example, suppose its CSP process is $B = q?x;\ Consume(x);\ B$. Thus B repeatedly receives an element along port q, stores it in a local variable x and consumes it.

Next, the process for the multi-capsule has to be constructed. Let us name this process MB (for $MultiB$). The semantics of a multi-capsule is the interleaving of all its instances. To achieve a correct addressing of the instances, communication over port q in the instance in is renamed into communication over $q.in$:

$$MB = \left|\left|\left|_{in:\{u,v,w\}} B[q \mapsto q.in]\right.\right.\right.$$

Thus the instance name in is transmitted as part of the value over channel q. For the above example B we obtain

$$B[q \mapsto q.in] = q.in?x;\ Consume(x);\ B[q \mapsto q.in].$$

Finally, compound capsule $System$ is constructed. This requires an instantiation of the address parameter of A with the set of receivers $\{u, v, w\}$, the instance names of the multi-capsule attached to port p.

$$System = \big(A(\{u, v, w\})[p \mapsto c]\ {}_{\{c\}}||_{\{c\}}\ MB[q \mapsto c]\big) \setminus \{c\}$$

Capsule A may now use the instance names u, v, w as parameters for channel p and thus send messages to particular instances.

3.3 Translation in General

Now we present a function $\mathcal{T}$ for translating a given capsule into a CSP process in equational form. We start from the basic types

$$[Process, Chans, Val]$$

of CSP processes, CSP channels, and values sent along channels. We assume a structure on the set of $Events$ based on a recursive free type $Data$ (defined in the style of Z):

$$Data ::= basic\langle\langle Val\rangle\rangle \mid comp\langle\langle InstName \times Data\rangle\rangle$$
$$Events == Chans \times Data$$

By convention, we abbreviate data of the form $comp(in, v)$ by $in.v$ and data of the form $basic(v)$ by just v. Also, following CSP conventions, events of the form (ch, d) are written as $ch.d$ so that a typical event will appear as $ch.in_1.....in_m.v$ where $m \in \mathbb{N}$. This dot notation for CSP communications should not be confused with the selection of components in Z schemas. Here the sequence $in_1.....in_m$ of instance names will play the role of addresses to where the value v should be sent along the channel ch.

With every process there is an *alphabet* of events associated:

$$\alpha : Process \to \mathbb{P} Events$$

The translation function

$$\mathcal{T} : Capsule \to ProcessEquations$$

generates a set of process equations of the form

$$name(ParameterList) = ProcessExpression$$

This set is defined inductively on the syntactic structure of capsules. Instead of writing it in a set notation we list the process equations one after another.

(1) Let BC be a basic capsule with $BC.ports = \{p_1, ..., p_m, q_1, ..., q_n\}$ where $m, n \in \mathbb{N}$ and $p_1, ..., p_m$ are single ports, i.e. with $p_i.multi = 1$ for $i = 1, ...m$, and $q_1, ..., q_n$ are multi-ports, i.e. with $q_j.multi > 1$ for $j = 1, ..., n$. Then we take new formal parameters $Adr_1, ..., Adr_n$ standing for sets of instance names that will serve as addresses to which the multi-ports $q_1, ..., q_n$ can be connected.
The translation function $\mathcal{T}$ generates one process equation $\mathcal{T}(BC)$ for BC:

$$\mathcal{T}(BC) \equiv BC.name(Adr_1, ..., Adr_n) = RHS$$

Here $\equiv$ stands for syntactic identity and RHS for a process with alphabet

$$\alpha(RHS) \subseteq (\{p_1, ..., p_m\} \times Data) \cup (\{q_1\} \times comp(Adr_1 \times Data) \cup ... \cup \{q_n\} \times comp(Adr_n \times Data)).$$

In our setting, RHS is the name of the corresponding CSP-OZ class.

(2) Let CC be a compound capsule with $CC.ports = \{p_1, ..., p_m, q_1, ..., q_n\}$ where $m, n \in \mathbb{N}$ and $p_1, ..., p_m, q_1, ..., q_n$ are as above. Then we take new formal parameters $Adr_1, ..., Adr_n$ as above and define

$$\mathcal{T}(CC) \equiv$$
$$CC.name(Adr_1, ..., Adr_n) = \qquad \text{[Eqn 1]}$$
$$\Big(\big(\ \|_{SN:CC.scnames}[\mathcal{C}(SN).ports[R_{SN,CC}]] \bullet SN(B_1, ..., B_{k(SN)})[R_{SN,CC}]\big)[P_{CC}]\Big) \setminus H_{CC}$$

$$\mathcal{T}(\mathcal{C}(SN)) \quad \text{for all } SN : CC.scnames \qquad \text{[Eqns 2]}$$

Thus $\mathcal{T}(CC)$ generates one new process equation (Eqn 1) where the right-hand side uses an iterated *alphabetised parallel* composition and adds to it the equations (Eqns 2) obtained by applying the translation function $\mathcal{T}$ inductively to all subcapsules in CC. The alphabetised parallel iterates over all names SN of subcapsules in CC.
Suppose $r_1, ..., r_{k(SN)}$ are the multi-ports of the subcapsule with the name SN. Then the actual address parameters B_i with $i = 1, ..., k(SN)$ for these multi-ports are defined as follows:

- $B_i = Adr_{p_j}$ if r_i is connected to a public multi-port p_j of CC
- $B_i = MC.inames$ if r_i is connected to a single port p_j of a multi-capsule MC inside CC

The renaming $R_{SN,CC}$ changes port names to connector names:

$$R_{SN,CC} = \{pn : PortName;\ cn : ConName \mid \exists\, p : \mathcal{C}(SN).ports;\ c : CC.conn \bullet p \in c.ports \land pn = p.name \land cn = c.name \bullet (pn \mapsto cn)\}$$

The renaming P_{CC} is used to rename connectors from a subcapsule to a public port on the border of the capsule back to the name of the public port:

$$P_{CC} = \{cn : ConName;\ pn : PortName \mid \exists\, p : CC.ports;\ c : CC.conn \bullet p \in c.ports \land pn = p.name \land cn = c.name \bullet (cn \mapsto pn)\}$$

Finally, all remaining connector names are hidden:

$$H_{CC} = \{c : CC.conn \bullet c.name\}$$

(3) Let MC be a multi-capsule with

$$\mathcal{C}(MC.cname).ports = \{p_1, ..., p_m, q_1, ..., q_n\}$$

where $m, n \in \mathbb{N}$ and $p_1, ..., p_m, q_1, ..., q_n$ are as above. Again we take new formal parameters $Adr_1, ..., Adr_n$ as above and define

$$\mathcal{T}(MC) \equiv$$

$$MC.name(Adr_1, ..., Adr_n) = \Big|\Big|\Big|_{in:MC.inames} MC.cname(Adr_1, ..., Adr_n)[\![in]\!] \qquad \text{[Eqn 1]}$$

$$\mathcal{T}(\mathcal{C}(MC.cname))) \qquad \text{[Eqn 2]}$$

Thus $\mathcal{T}(MC)$ generates one new process equation (Eqn 1) where the right-hand side uses an iterated *interleaving* operator and adds to it the equation (Eqn 2) obtained by applying the translation function $\mathcal{T}$ inductively to the capsule inside MC. The interleaving operator iterates over all instances of this capsule.
Each *instance* is formalised by applying a renaming operator denoted by the postfix $\_[\![in]\!]$ where in is an instance name. For a given process P this operator is defined by

$$P[\![in]\!] \equiv P[\{ch : Chans;\ d : Data \mid ch.d \in \alpha(P) \bullet ch.d \mapsto ch.in.d\}]$$

(cf. the construction of the process MB in the example in subsection 3.2).

In the following we assume that a collaboration diagram implicitly contains a capsule *System* enclosing all capsules appearing in the diagram. The CSP process for the capsule *System* gives the architecture description.

4 Case Study: Automatic Manufacturing System

In this section we apply our approach to a larger case study. It concerns the architectural description of an *automatic manufacturing system*. In automatic manufacturing systems the transportation of material between machine tools is carried out by autonomous or *holonic* transportation agents, i.e. vehicles or robots without drivers and without a central control for scheduling[2]. In [19] a CSP-OZ specification of an automatic manufacturing system is given in which the architecture of the system is described by a CSP term and the components (capsules) are given by CSP-OZ class definitions. Here, we will only present the architecture, initially modelled by a UML-RT collaboration diagram and then translated into a CSP term using the translation $\mathcal{T}$ of the previous section.

The automatic manufacturing system consists of the following parts (see Figure 3): two stores *In* and *Out*, one for workpieces to be processed (the in-store) and one for the finished workpieces (the out-store); two holonic transportation systems (Hts) $T1$ and $T2$; and three machine tools (Wzm[3]) A, B and C for processing the workpieces. Every workpiece has to be processed by all three

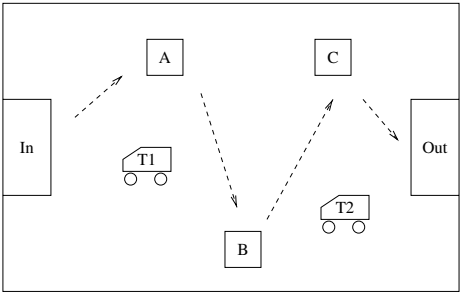


Fig. 3. Plant

machine tools in a fixed order ($In \rightarrow A \rightarrow B \rightarrow C \rightarrow Out$). The Hts' are responsible for transporting the workpieces between machine tools and stores. They work as autonomous agents, free to decide which machine to serve (within some chosen strategy). Initially the in-store is full and the out-store as well as all machine tools are empty. When a machine is empty or contains an already processed workpiece it broadcasts a *request* to the Hts in order to receive a new workpiece or to deliver one. The Hts' (when listening) send some *offer* to the machine tools, telling them their cost for satisfying the request. Upon receipt of offers the machine decides for the best offer and give this Hts the *order*, which then executes it. Execution of a job involves *loading* and *deloading* of workpieces

[2] This case study is part of the priority research program "Integration of specification techniques with applications in engineering" of the German Research Council (DFG) (`http://tfs.cs.tu-berlin.de/projekte/indspec/SPP/index.html`).

[3] in German: Werkzeugmaschine

from/to Hts and from/to stores and machine tools. This way, all workpieces are processed by all three tools and transported from the in- to the out-store.

The CSP-OZ specification of this manufacturing system contains class definitions *Store* and *Wzm*. The most complex component *Hts* is split into three parts: one for managing the acquisition of new jobs (*Acquisition*), one for coordinating the driving in the plant hall (*Driver*) and a control part (*HtsCtrl*). The control part calls component *Acquisition* when an order should be *acquired* (with response *new order*), and component *Driver* when the vehicle has to move. Furthermore, component *Acquisition* frequently asks *Driver* about their current position, which is influencing the cost of offers. Figure 4 shows the architecture

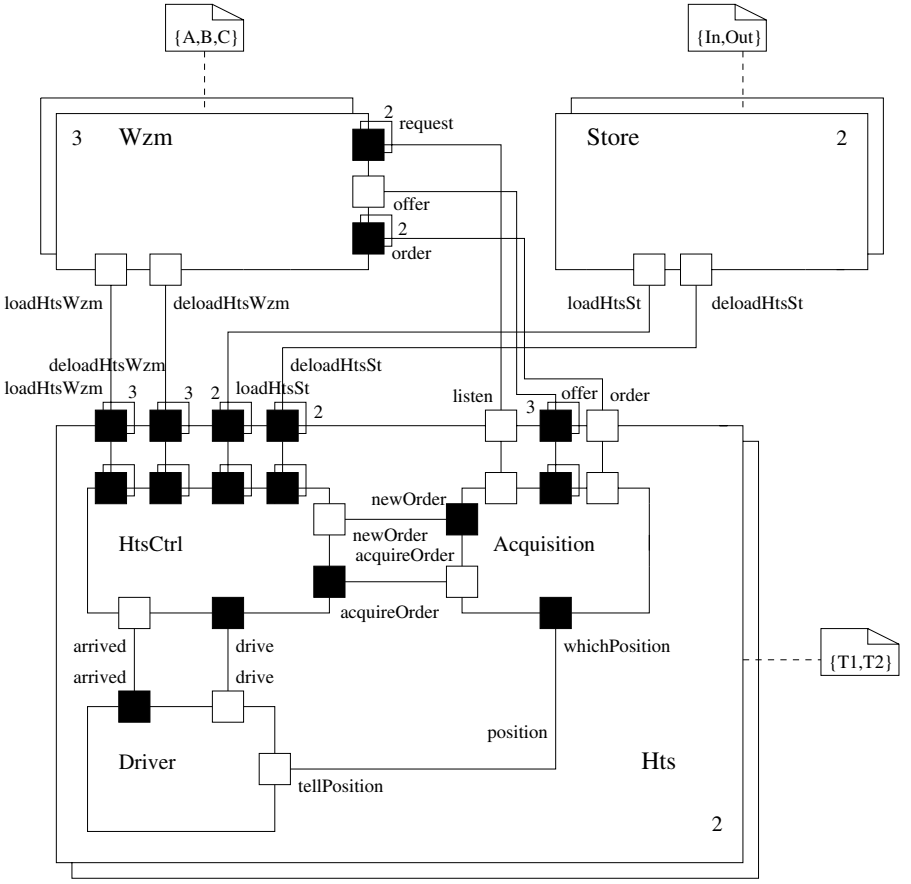


Fig. 4. Architecture of the Manufacturing System

of the manufacturing system as an UML-RT structure diagram. In most cases we have omitted the names of connectors. They only appear at the places where

they are necessary in the transformation: when the connected ports have different names. In the other cases, we simply assume that the connector name equals the names of connected ports (and consequently omit the otherwise necessary renaming). We also sometimes omit port names of subcapsules when they agree with the port names of the compound capsule. The set of process equations corresponding to the graphical architecture description is constructed inductively. First, the process names and parameters for all basic capsules are fixed. For each of these names, a corresponding CSP-OZ class has to be declared in the rest of the specification. Thus we for instance get classes $Acquisition(Adr_{offer})$ and $HtsCtrl(Adr_{loadHtsWzm}, Adr_{deloadHtsWzm}, Adr_{loadHtsSt}, Adr_{deloadHtsSt})$.

Next, we construct the process equations for the multi-capsules *Store* and *Wzm*. Their process names are *MStore* and *MWzm*, respectively. Due to lack of space we omit them here and focus on the most complex component *Hts*.

First, the process equation for the compound capsule is constructed.

$$
\begin{aligned}
&Hts(Adr_{loadHtsWzm}, Adr_{deloadHtsWzm},\\
&\quad Adr_{loadHtsSt}, Adr_{deloadHtsSt}, Adr_{offer}) =\\
&\quad \big(Driver[tellPosition \mapsto position]\ {}_{A_{Driver}}||_{A_{Acqui} \cup A_{Ctrl}}\\
&\qquad \big(Acquisition(Adr_{offer})[whichPosition \mapsto position]\ {}_{A_{Acqui}}||_{A_{Ctrl}}\\
&\qquad\ HtsCtrl(Adr_{loadHtsWzm}, Adr_{deloadHtsWzm}, Adr_{loadHtsSt}, Adr_{deloadHtsSt})\big)\big)\\
&\qquad\qquad \setminus H_{Hts}
\end{aligned}
$$

For reasons of readability we have unfolded the iterated alphabetised parallel composition in the process equation for *Hts*. The sets $A_{...}$ contain all names of connectors attached to the capsule (in this case equal to the corresponding port names), $H_{Hts} = \{arrived, drive, position, acquireOrder, newOrder\}$. The renaming P_{Hts} is in this case empty since we have adopted the convention that omitted connector names equal their port names.

Next, the process equation for the multi-capsule *MHts* is constructed:

$$
\begin{aligned}
&MHts(Adr_{loadHtsWzm}, Adr_{deloadHtsWzm},\\
&\quad Adr_{loadHtsSt}, Adr_{deloadHtsSt}, Adr_{offer}) =\\
&\quad |||_{in:\{T1,T2\}} \bullet Hts(Adr_{loadHtsWzm}, Adr_{deloadHtsWzm},\\
&\qquad\qquad Adr_{loadHtsSt}, Adr_{deloadHtsSt}, Adr_{offer})[\![in]\!]
\end{aligned}
$$

Finally we can give the system description, applying once again the translation scheme for compound capsules:

$$
\begin{aligned}
&System =\\
&\quad \big(MHts(\{A,B,C\},\{A,B,C\},\{In,Out\},\{In,Out\},\{A,B,C\})\\
&\qquad {}_{A_{MHts}}||_{A_{MWzm} \cup A_{MStore}}\\
&\qquad \big(MWzm(\{T1,T2\},\{T1,T2\})\ {}_{A_{MWzm}}||_{A_{MStore}}\ MStore\big)\big) \setminus H_{System}
\end{aligned}
$$

This completes the translation.

5 Conclusion

In this paper, we have proposed a translation of UML-RT structure diagrams into CSP. This allows us to use graphical architecture descriptions in CSP-OZ without losing its formal semantics. The technique is not only applicable to CSP-OZ specifications, but more generally to all specification languages which use CSP for structure descriptions. The only change needed then is the interpretation of basic capsules, which stand for CSP-OZ classes in our case but may also be interpreted differently. The translation gives *one* possible semantics to UML-RT collaboration diagrams. In a different setting (for instance hardware design on a much lower abstraction level), a different semantics might be conceivable.

The basis for the translation given in this paper is a formalisation of the syntax of UML-RT structure diagrams in Object-Z. This is similar to the work done in [10], which formalises the syntax of UML class diagrams with Z and uses the formalisation for a translation of class diagrams to Object-Z classes.

So far we have not explicitly treated *protocols*, which are also part of UML-RT. Protocols are used for specifying the type of interactions which may take place over some connector. A protocol can for instance define a set of signals passed over a connector, or can give a valid communication sequence. Since the basic communication paradigm of CSP is *synchronous* communication, we have assumed that all protocols define synchronous communication over a single channel. However, we envisage the possibility of using more elaborate protocols in an architecture description, for instance protocols for defining asynchronous communication or communication over unreliable channels. To fit into the CSP view on UML-RT diagrams, these protocols should be specified in CSP. This approach to protocol definition is similar to the method chosen in WRIGHT [1], an architecture description language (ADL) based on CSP. In WRIGHT, an architecture description consists of a set of components together with a collection of *connectors* which are given in CSP.

Another work similar to ours is the ADL Darwin [11]. Darwin is both a graphical and a textual modelling language; it has a formal semantics in terms of Milner's π-calculus [12]. The usage of a calculus with mobility is necessary there because Darwin allows to specify dynamically evolving system structures. Since our goal was to find a graphical description of *CSP* structure specifications, we had no need for incorporating facilities for describing mobility.

A completely different semantic approach to UML-RT diagrams can be found in [8]. There the focus is on using a visual but still well-defined formalism (*interaction graphs*) for interpreting UML-RT diagrams.

References

1. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 1997.
2. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language user guide*. Addison Wesley, 1999.

3. T. Clark and A. Evans. Foundations of the unified modeling language. In *Northern Formal Methods Workshop*, Electronic Workshops in Computing. Springer, 1998.
4. S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. Translating the OMT dynamic model into Object-Z. In J.P. Bowen, A. Fett, and M.G. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *Lecture Notes in Computer Science*, pages 347–366. Springer, 1998.
5. C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, volume 2, pages 423–438. Chapman & Hall, 1997.
6. C. Fischer. *Combination and Implementation of Processes and Data: From CSP-OZ to Java*. PhD thesis, Bericht Nr. 2/2000, University of Oldenburg, April 2000.
7. R. France and B. Rumpe, editors. *UML'99: The Modified Modeling Language – Beyond the Standard*, volume 1723 of *Lecture Notes in Computer Science*. Springer, 1999.
8. R. Grosu, M. Broy, B. Selic, and G. Stefanescu. What is behind UML-RT? In *Behavioural Specifications of business and systems*. Kluwer, 1999.
9. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
10. S.-K. Kim and D. Carrington. Formalizing the UML class diagram using Object-Z. In R. France and B. Rumpe, editors, *UML'99: The Unified Modelling Language – Beyond the Standard*, volume 1723 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 1999.
11. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *ESEC '95: European Software Engineering Conference*, 1995.
12. R. Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, 1999.
13. Object Management Group. *OMG Unified Modeling Language Specification*, June 1999. version 1.3.
14. E.-R. Olderog and A.P. Ravn. Documenting design refinement. In M.P.E. Heimdahl, editor, *Proc. of the Third Workshop on Formal Methods in Software Practice*, pages 89–100. ACM, 2000.
15. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
16. B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. Technical report, ObjecTime, 1998.
17. G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publisher, 2000.
18. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International Series in Computer Science, 2nd edition, 1992.
19. H. Wehrheim. Specification of an automatic manufacturing system – a case study in using integrated formal methods. In T. Maibaum, editor, *FASE 2000: Fundamental Aspects of Software Engineering*, number 1783 in LNCS. Springer, 2000.

Strengthening UML Collaboration Diagrams by State Transformations*

Reiko Heckel and Stefan Sauer

University of Paderborn, Dept. of Mathematics and Computer Science
D-33095 Paderborn, Germany
reiko|sauer@uni-paderborn.de

Abstract. Collaboration diagrams as described in the official UML documents specify patterns of system structure and interaction. In this paper, we propose their use for specifying, in addition, pre/postconditions and state transformations of operations and scenarios. This conceptual idea is formalized by means of graph transformation systems and graph process, thereby integrating the state transformation with the structural and the interaction aspect.

Keywords: UML collaboration diagrams, pre/postconditions, graph transformation, graph process

1 Introduction

The Unified Modeling Language (UML) [24] provides a collection of loosely coupled diagram languages for specifying models of software systems on all levels of abstraction, ranging from high-level requirement specifications over analysis and design models to visual programs. On each level, several kinds of diagrams are available to specify different aspects of the system, like the structural, functional, or interaction aspect. But even diagrams of the same kind may have different interpretations when used on different levels, while several aspects of the same level may be expressed within a single diagram.

For example, interaction diagrams in UML, like sequence or collaboration diagrams, often represent sample communication scenarios, e.g., as refinement of a use case, or they may be used in order to give a complete specification of the protocol which governs the communication. Collaboration diagrams allow, in addition, to represent individual snapshots of the system as well as structural patterns.

If such multi-purpose diagrammatic notations shall be employed successfully, a precise understanding of their different aspects and abstraction levels is required, as well as a careful analysis of their mutual relations. This understanding, once suitably formalized, can be the basis for tool support of process models, e.g., in the form of consistency checks or refinement rules.

* Research partially supported by the ESPRIT Working Group APPLIGRAPH.

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 109–123, 2001.

In this paper, we address these issues for UML collaboration diagrams. These diagrams are used on two different levels, the instance and the specification level, both related to a class diagram for typing (cf. Fig. 1). A specification-level diagram provides a pattern which may occur at the instance level.[1]

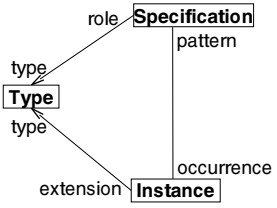


Fig. 1. Two levels of collaboration diagrams and their typing

In addition, in the UML specification [24] two aspects of collaboration diagrams are identified: the *structural aspect* given by the graph of the collaboration, and the *interaction aspect* represented by the flow of messages. These aspects are orthogonal to the dimensions in Fig. 1: A specification-level diagram may provide a structural pattern as well as a pattern of interaction. At the instance level, a collaboration diagram may represent a snapshot of the system or a sample interaction scenario. Moreover, both aspects are typed over the class diagram, and the pattern-occurrence relation should respect this typing.

One way to make precise the relationships between different diagrams and abstraction levels is the approach of *meta modeling* used in the UML specification [24]. It allows to specify the syntactic relation between different diagrams (or different uses of the same diagram) by representing the entire model by a single *abstract syntax graph* where dependencies between different diagrams can be expressed by means of additional links, subject to structural constraints specifying consistency. This approach provides a convenient and powerful language for integrating diagram languages, i.e., it contributes to the question, *how* the integration can be specified. However, it provides no guidelines, *what* the intended relationships between different diagrams should be.

Therefore, in this paper, we take the alternative approach of translating the diagrams of interest into a formal method which is conceptually close enough in order to provide us with the required semantic intuition to answer the *what* question. Once this is sufficiently understood, the next step is to formulate these results in the language of the UML meta model.

Our formal method of choice are graph transformation systems of the so-called *algebraic double-pushout (*DPO*) approach* [10] (see [5] for a recent survey).

[1] The use of collaboration diagrams for *role modeling* is not captured by this picture. A role model provides a refinement of a class diagram where roles restrict the features of classes to those relevant to a particular interaction. A collaboration diagram representing a role model can be seen as a second level of typing for instance (and specification-level) diagrams which itself is typed over the class diagram. For simplicity, herein we restrict ourselves to a single level of typing.

In particular, their typed variant [4] has built in most of the aspects discussed above, including the distinction between pattern, instance, and typing, the structural aspect and (by way of the partial order semantics of *graph processes* [4]) a truly concurrent model for the interaction aspect. The latter is in line with the recent proposal for UML action semantics [1] which identifies a semantic domain for the UML based on a concurrent data flow model.

The direct interpretation of class and collaboration diagrams as graphs and of their interrelations as graph homomorphisms limits somewhat the scope of the formalization. In particular, we deliberately neglect inheritance, ordered or qualified associations, aggregation, and composition in class diagrams as well as multi-objects in collaboration diagrams. This oversimplification for presentation purpose does not imply a general limitation of the approach as we could easily extend the graph model in order to accommodate these features, e.g., using a meta model-based approach like in [21].

Along with the semantic intuition gained through the interpretation of collaboration diagrams in terms of graph transformation comes a conceptual improvement: the use of collaboration diagrams as a visual query and update language for object structures. In fact, in addition to system structure and interaction, we propose the use of collaboration diagrams for specifying the *state transformation* aspect of the system. So far, this aspect has been largely neglected in the standard documents [24], although collaboration diagrams are used already in the CATALYSIS approach [6] and the FUSION method [3] for describing pre- and postconditions of operations and scenarios.

Beside a variety of theoretical studies, in particular in the area of concurrency and distributed systems [9], application-oriented graph transformation approaches like PROGRES [29] or FUJABA [14] provide a rich background in using rule-based graph transformation for system modeling as well as for testing, code generation, and rapid prototyping of models (see [7] for a collection of survey articles on this subject). Recently, graph transformations have been applied to UML meta modeling, e.g., in [16,2,11].

Therefore, we believe that our approach not only provides a clarification, but also a conceptual improvement of the basic concepts of collaboration diagrams.

Two approaches which share the overall motivation of this work remain to be discussed, although we do not formally relate them herein. Övergaard [26] uses sequences in order to describe the semantics of interactions, including notions of refinement and the relation with use cases. The semantic intuition comes from trace-based interleaving models which are popular, e.g., in process algebra. Knapp [22] provides a formalization of interactions using temporal logic and the *pomset* (partially ordered multi-set) model of concurrency [27]. In particular, the pomset model provides a semantic framework which has much similarity with graph processes, only that pomsets are essentially set-based while graph processes are about graphs, i.e., the structural aspect is already built in. Besides technical and methodological differences with the cited approaches, the main additional objective of this work is to *strengthen collaboration diagrams by incorporating graph transformation concepts.*

The presentation is structured according to the three aspects of collaboration diagrams. After introducing the basic concepts and a running example in Sect. 2, Sect. 3 deals with the structural and the transformation aspect, while Sect. 4 is concerned with interactions. Section 5 concludes the paper.

A preliminary sketch of the ideas of this paper has been presented in [20].

2 UML Collaboration Diagrams: A Motivating Example

In this section, we introduce a running example to motivate and illustrate the concepts in this paper. First, we sketch the use of collaboration diagrams as suggested in the UML specification [24]. Then, we present an improved version of the same example exploiting the state transformation aspect.

Figure 2 shows the class diagram of a sample application where a Company object is related to zero or more Store, Order, and Delivery objects. Order objects as well as Delivery objects are related to exactly one Customer who plays the role of the customer placing the order or the receiver of a delivery, respectively. A Customer can place several instances of Order and receive an unrestricted number of Delivery objects.

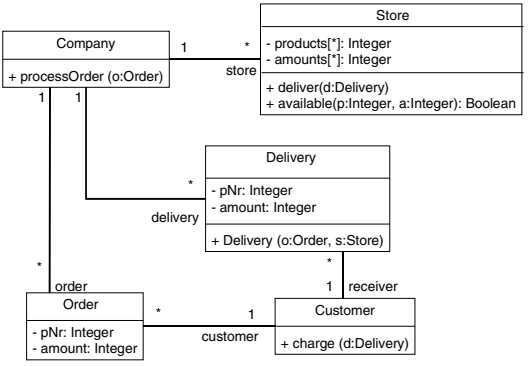


Fig. 2. A class diagram defining the structure of the example

A typical scenario within that setting is the situation where a customer orders a product from the company. After the step of refining and combining different use cases into a method-oriented specification one might end up with a collaboration diagram specifying the implementation of operation processOrder as depicted in the top of Fig. 3. Here, the company first obtains the product number pNr and the ordered amount using defined access functions. It then checks all stores to find one that can supply the requested amount of the demanded product. A delivery is created, and the selected store is called to send it out. Concurrently, the customer is charged for this delivery. After an order has been processed, it will be deleted.

Collaboration diagrams like this, which specifies the execution of an operation, may be used for generating method implementations in Java [12], i.e., they

can be seen as visual representations of programs. However, in earlier phases of development, a higher-level style of specification is desirable which abstracts from implementation details like the get functions for accessing attributes and the implementation of queries by search functions on multi-objects.

Therefore, we propose to interpret a collaboration as a visual query which uses pattern matching on objects, links, and attributes instead of low-level access and search operations. In fact, leaving out these details, the same operation can be specified more abstractly by the diagram in the lower left of Fig. 3. Here, the calls to getpNr and getAmount are replaced by variables p and a for the corresponding attribute values, and the call of search on the multi-object is replaced by a boolean function available which constrains the instantiation of /s:Store. (As specified in the lower right of the same figure, the function returns true if the Store object matching /s is connected to a Product object with the required product number p and an amount b greater than a.) The match is complete if all items in the diagram not marked as {new} are instantiated. Then, objects marked as {destroyed} are removed from the current state while objects marked as {new} are created, initializing appropriately the attributes and links. For example, the new Delivery object inherits its link and attribute values from the destroyed Order object.

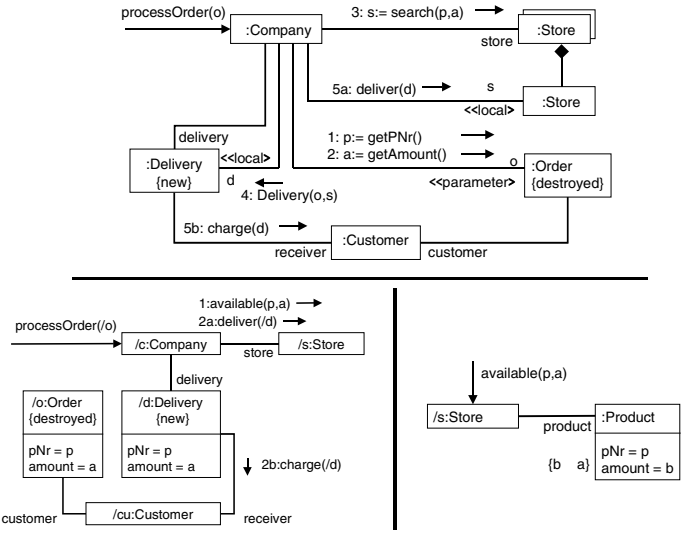


Fig. 3. An implementation-oriented collaboration diagram (top), its declarative presentation (bottom left), and a visual query operation (bottom right)

In the following sections, we show how this more abstract use of collaboration diagrams can be formalized by means of graph transformation rules and graph processes.

3 Collaborations as Graph Transformations

A collaboration on specification level is a graph of classifier roles and association roles which specifies a view of the classes and associations of a class diagram as well as a pattern for objects and links on the instance level. This triangular relationship, which instantiates the type-specification-instance pattern of Fig. 1 for the structural aspect, shall be formalized in the first part of this section. Then, the state transformation aspect shall be described by means of graph transformations. The interaction aspect is considered in the next section.

Structure. Focusing on the structural aspect first, we use graphs and graph homomorphisms (i.e., structure-compatible mappings between graphs) to describe the interrelations between class diagrams and collaboration diagrams on the specification and the instance level.

The relation between class and instance diagrams is formally captured by the concept of *type* and *instance graphs* [4]. By *graphs* we mean directed unlabeled graphs $G = \langle G_V, G_E, src^G, tar^G \rangle$ with set of vertices G_V, set of edges G_E, and functions $src^G : G_E \to G_V$ and $tar^G : G_E \to G_V$ associating to each edge its source and target vertex, respectively. A graph homomorphism $f : G \to H$ is a pair of functions $\langle f_V : G_V \to H_V, f_E : G_E \to H_E \rangle$ compatible with source and target, i.e., for all edges e in G_E, $f_V(src^G(e)) = src^H(f_E(e))$ and $f_V(tar^G(e)) = tar^H(f_E(e))$.

Let TG be the underlying graph of a class diagram, called *type graph*. A legal *instance graph* over TG consists of a graph G together with a typing homomorphism $g : G \to TG$ associating to each vertex and edge x of G its type $g(x) = t$ in TG. In UML notation, we write $x : t$. Observe that the compatibility of g with source and target ensures that, e.g., the class of the source object of a link is the source class of the link's association. Constraints like this can be found in the meta class diagrams and well-formedness rules of the UML meta model for the meta associations relating classifiers with instances, associations with links, association ends with link ends, etc. ([24], Sect. 2.9). The typing of specification-level graphs is described in a similar way ([24], Sect. 2.10).

An interpretation of a graph homomorphism which is conceptually different, but requires the same notion of structural compatibility, is the occurrence of a pattern in a graph. For example, a collaboration on the specification level occurs in a collaboration on the instance level if there exists a mapping from classifier roles to instances and from association roles to links preserving the connections. Thus, the existence of a graph homomorphism from a given pattern graph implies the presence of a corresponding instance level structure. The occurrence has to be type-compatible, i.e., if a classifier role is mapped to an instance, both have to be of the same classifier. This compatibility is captured in the notion of a *typed graph homomorphism* between typed graphs, i.e., a graph homomorphism which preserves the typing. In our collaboration diagrams, this concept of graphical pattern matching is used to express visual queries on object structures.

In summary, class and collaboration diagrams have a homogeneous, graph-like structure, and their triangular relationship can be expressed by three compatible graph homomorphisms. Next, this triangular relation shall be lifted to the state transformation view.

State Transformation. Collaborations specifying queries and updates of object structures are formalized as *graph transformation rules*, while corresponding collaborations on the instance level represent individual *graph transformations*.

A *graph transformation rule* $r = L \to R$ consists of two graphs L, R such that the union $L \cup R$ is defined. (This ensures that, e.g., edges which appear in both L and R are connected to the same vertices in both graphs.) Consider the rule in the upper part of Fig. 4 representing the collaboration of processOrder in the lower left of Fig. 3. The precondition L contains all objects and links which have to be present before the operation, i.e., all elements of the diagram except for /d:Delivery which is marked as {new}. Analogously, the postcondition R contains all elements except for /o:Order which is marked as {destroyed}. (The {transient} constraint does not occur because a graph transformation rule is supposed to be atomic, i.e., conceptually there are no intermediate states between L and R.)

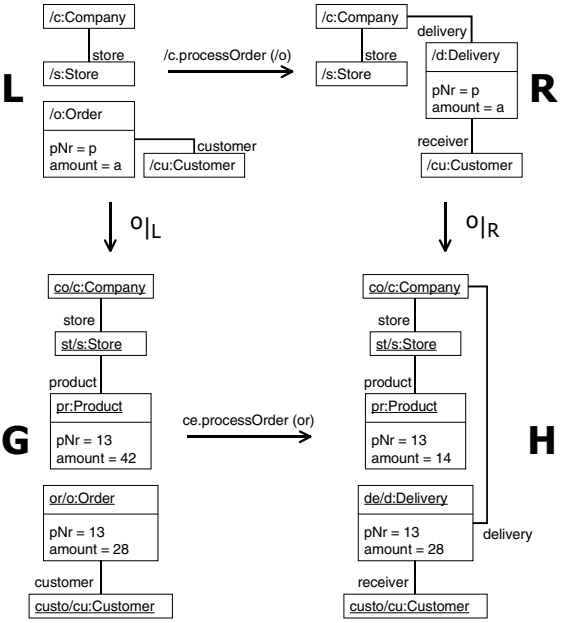


Fig. 4. A graph transition consisting of a rule $L \to R$ specifying the operation processOrder (top), and its occurrence o in an instance-level transformation (bottom)

A similar diagram on the instance level represents a graph transformation. Graph transformation rules can be used to specify transformations in two different ways: either operationally by requiring that the rule is applied to a given graph in order to rewrite part of it, or axiomatically by specifying pre- and postconditions. In the first interpretation (in fact, the classical one [10], in set-theoretic formulation), a *graph transformation* $G \overset{r(o)}{\Longrightarrow} H$ from a pre-state G to a post-state H using rule r is represented by a graph homomorphism $o : L \cup R \to G \cup H$, called *occurrence*, such that

1. $o(L) \subseteq G$ and $o(R) \subseteq H$ (i.e., the left-hand side of the rule is matched by the pre-state and the right-hand side by the post-state),
2. $o(L \setminus R) = G \setminus H$ and $o(R \setminus L) = H \setminus G$ (i.e., all objects of G are $\{\mathsf{destroyed}\}$ that match classifier roles of L not belonging to R and, symmetrically, all objects of H are $\{\mathsf{new}\}$ that match classifier roles in R not belonging to L).

That is, the transformation creates and destroys exactly what is specified by the rule and the occurrence. As a consequence, the rule together with the occurrence of the left-hand side L in the given graph G determines, up to renaming, the derived graph H, i.e., the approach has a clear operational interpretation, which is well-suited for visual programming.

In the more liberal, axiomatic interpretation, requirement 2 is weakened to

$2'$. $o(L \setminus R) \subseteq G \setminus H$ and $o(R \setminus L) \subseteq H \setminus G$ (i.e., *at least* the objects of G are $\{\mathsf{destroyed}\}$ that match classifier roles of L not belonging to R and, symmetrically, *at least* the objects of H are $\{\mathsf{new}\}$ that match classifier roles in R not belonging to L).

These so-called *graph transitions* [19] allow side effects not specified by the rule, like in the example of Fig. 4 where the amount of product pr changes without being explicitly rewritten by the rule. This is important for high-level modeling where specifications of behavior are often incomplete.

In both cases, instance transformations as well as specification-level rules are typed over the same type graph, and the occurrence homomorphism respects these types. This completes the instantiation of the type-specification-instance pattern for the aspect of state transformation.

Summarizing, a collaboration on the specification level represents a pattern for state transformations on the instance level, and the occurrence of this pattern requires, beside the structural match of the pre- and postconditions, (at least) the realization of the described effects. Graph transformations provide a formal model for the state transformation aspect which allows to describe the overall effect of a complex interaction. However, the interaction itself, which decomposes the global steps into more basic actions, is not modeled. In the next section, this finer structure shall be described in terms of the model of concurrency for graph transformation systems [4].

4 Interactions as Graph Processes

In this section, we shall extend the triangular type-specification-instance pattern to the interaction part. First, we describe the formalization of the individual concepts and then the typing and occurrence relations.

Class diagrams. A class diagram is represented as a *graph transformation system*, briefly GTS, $\mathcal{G} = \langle TG, \mathcal{R} \rangle$ consisting of a type graph TG and a set of transformation rules $\mathcal{R}$. The type graph captures the structural aspect of the class diagram, like the classes, associations, and attributes, as well as the types of call and return messages that are sent when an operation is invoked. For the

fragment of the class diagram consisting of the classes Order, Customer, and Delivery, the customer association[2], and the attributes and operations of the first two classes, the type graph is shown in Fig. 5. Classes are as usually shown as rectangular, data types as oval shapes. Call and return messages are depicted like UML action states, i.e., nodes with convex borders at the two sides. Return messages are marked by overlined labels. Links from call message nodes represent the input parameters of the operation, while links from return message nodes point to output parameters (if any). The self link points to the object executing the operation. The rules of the GTS in Fig. 5 model different kinds of basic actions that are implicitly declared within the class diagram. Among them are state transformation actions like destroy o, control actions like send charge, and actions representing the execution of an operation like cu.charge(d).

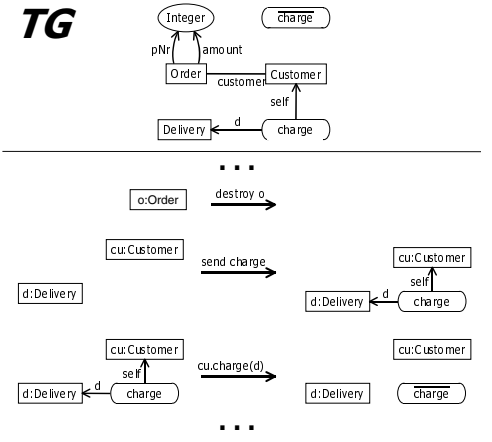


Fig. 5. Graph transformation system for a fragment of the class diagram in Fig. 2

Interactions. An interaction consists of a set of messages, linked by control and data flow, that stimulate actions like access to attributes, invocation of operations, creation and deletion of objects, etc. While the control flow is explicitly given by sequence numbers specifying a partial order over messages, data flow information is only implicitly present, e.g., in the parameters of operations and in their implementation, as far as it is given. However, it is important that control and data flow are compatible, i.e., they must not create cyclic dependencies. Such constraints are captured by the concept of a *graph process* which provides a partial order semantics for graph transformation systems.

The general idea of process semantics, which have their origin in the theory of Petri nets [28], is to abstract, in an individual run, from the ordering of actions that are not causally dependent, i.e., which appear in this order only by accident or because of the strategy of a particular scheduler. If actions are represented

[2] More precisely, in UML terms this is an unnamed association in which class Customer plays the role customer.

by graph transformation rules specifying their behavior in a pre/postcondition style, these causal dependencies can be derived by analyzing the intersections of rules in the common context provided by the overall collaboration.

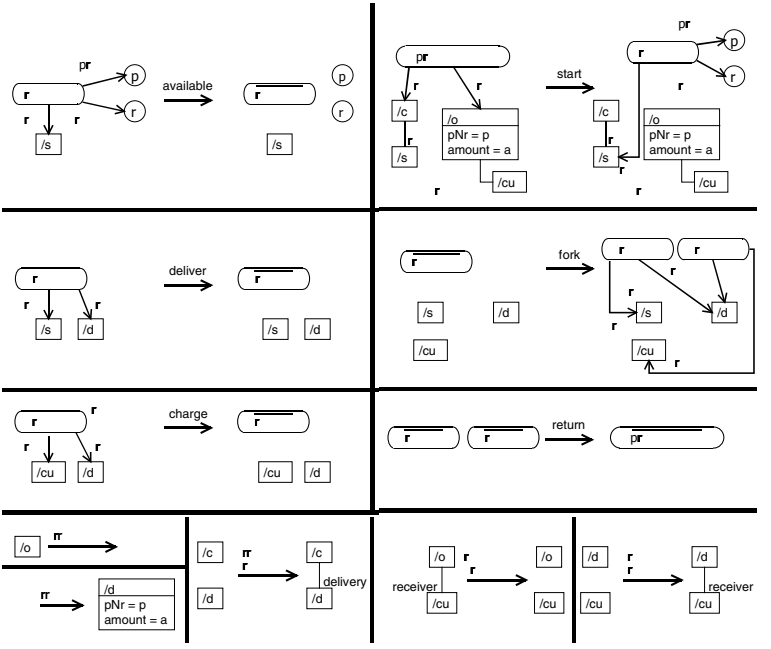


Fig. 6. Graph process for the collaboration diagram of operation processOrder. The three rules in the upper left section represent the operations, those in the upper right realize the control flow between these operations, and the five rules in the lower part are responsible for state transformations (note that attribute values a and p of /d can be instantiated by new /d since all the rules of the graph process act in a common context given by the collaboration)

The graph process for the collaboration diagram in the lower left of Fig. 3 is shown in Fig. 6. It consists of a set of rules representing the internal actions, placed in a common name space. That means, e.g., the available node created by the rule start in the top right is the same as the one deleted by the rule available in the top left. Because of this causal dependency, available has to be performed after start. Thus, the causality of actions in a process is represented by the overlapping of the left- and right-hand sides of the rules.

Graph processes are formally defined in three steps. A *safe graph transformation system* consists of a graph C (best to be thought of as the graph of the collaboration) together with a set of rules $\mathcal{T}$ such that, for every rule $t = G \to H \in \mathcal{T}$ we have $G, H \subseteq C$ (that is, C provides a common context for the rules in $\mathcal{T}$). Intuitively, the rules in $\mathcal{T}$ represent transformations, i.e., occurrences of rules. In order to formalize this intuition, the notion of *occurrence graph transformation system* is introduced requiring, in addition, that the

transformations in $\mathcal{T}$ can be ordered in a sequence. That means, the system has to be acyclic and free of conflicts, and the causality relation has to be compatible with the graphical structure. In order to make this precise, we define the causal relation associated to a safe GTS $\langle C, \mathcal{T} \rangle$. Let $t : G \to H$ be one arbitrary transformation in $\mathcal{T}$ and e be any edge, node, or attribute in C. We say that

- t *consumes* e if $e \in G \setminus H$
- t *creates* e if $e \in H \setminus G$
- t *preserves* e if $e \in G \cap H$

The relation $\leq$ is defined on $\mathcal{T} \cup C$, i.e., it relates both graphical elements and operations. It is the transitive and reflexive closure of the relation $<$ where

- $e < t_1$ if t_1 consumes e
- $t_1 < e$ if t_1 creates e
- $t_1 < t_2$ if t_1 creates e and t_2 preserves e, or
  t_1 preserves e and t_2 consumes e

Now, a safe GTS is called an *occurrence graph transformation system* if

- the causal relation $\leq$ is a partial order which respects source and target, i.e., if $t \in \mathcal{T}$ is a rule and a vertex v of C is source (or target) of an edge e, then
  - $t \leq v$ implies $t \leq e$ and
  - $v \leq t$ implies $e \leq t$
- for all elements x of C, x is consumed by at most one rule in $\mathcal{T}$, and it is created by at most one rule in $\mathcal{T}$.

The objective behind these conditions is to ensure that each occurrence GTS represents an equivalence class of sequences of transformations "up-to-rescheduling", which can be reconstructed as the linearizations of the partial order $\leq$. Vice versa, from each transformation sequence one can build an occurrence GTS by taking as context C the colimit (sum) of all instance graphs in the sequence [4].

The causal relation between the rules in the occurrence GTS in Fig. 6 is visualized by the Petri net in the left of Fig. 7. For example, the dependency between available and start discussed above is represented by the place between the corresponding transitions. Of these dependencies, which include both control- and data-flow, in the UML semantics only the control-flow dependencies are captured by a precedence relation on messages, specified using sequence numbers. This part is presented by the sub-net in the upper right of Fig. 7. In comparison, the net in the lower right of Fig. 7 visualizes the control flow if we replace the synchronous call to deliver by an asynchronous one: The call is delegated to a thread which consumes the return message and terminates afterwards. This strategy for modeling asynchronous calls might seem a little *ad-hoc*, but it follows the implementation of asynchronous calls in Java as well as the formalization of asynchronous message passing in process calculi [23]. The essential property is the independence of the deliver and the charge action.

We have used graph transformation rules for specifying both the overall effect of an interaction as well as its basic, internal actions. A fundamental fact about

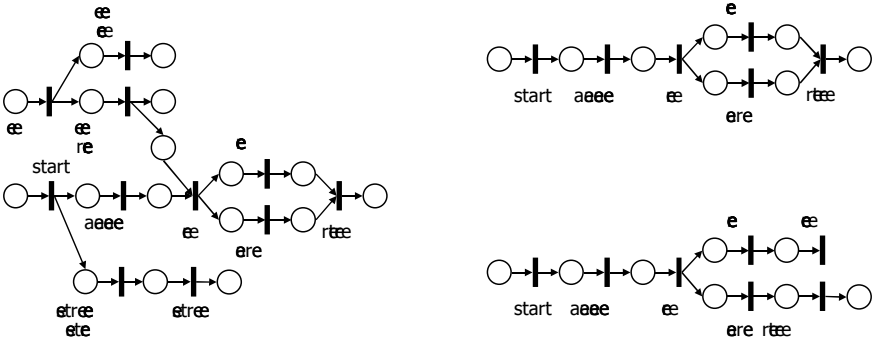


Fig. 7. Control flow and data dependencies of the occurrence GTS in Fig. 6 (left), sub-net for control flow dependencies of occurrence GTS (top right), and control flow dependencies with asynchronous call of deliver (bottom right)

occurrence GTS [4] relates the state transformation with the interaction aspect: Given an occurrence GTS $\mathcal{O} = \langle C, \mathcal{T} \rangle$ and its partial order $\leq$, the sets of minimal and maximal elements of C w.r.t. $\leq$ form two graphs $Min(\mathcal{O}), Max(\mathcal{O}) \subseteq C$. This allows us to view a process p externally as a transformation rule $\tau(p)$, called *total rule of* p, which combines the effects of all the local rules of $\mathcal{T}$ in a single, atomic step. The total rule of the process in Fig. 6 is shown in the top of Fig. 4.

Summarizing, the three corners of our triangle are represented by a GTS representing the type level, and two occurrence GTS formalizing interactions on the specification and the instance level, respectively. It remains to define the relation between these three levels, i.e., the concepts of typing and occurrence.

Typing. In analogy with the typing of graphs, an occurrence GTS $\mathcal{O} = \langle C, \mathcal{T} \rangle$ is typed over a GTS $\mathcal{G} = \langle TG, \mathcal{R} \rangle$ via a *homomorphism of graph transformation systems*, briefly GTS morphism. A GTS morphism $p : \mathcal{O} \to \mathcal{G}$ consists of a graph homomorphism $c : C \to TG$ typing the context graph C over the type graph TG, and a mapping of rules $f : \mathcal{T} \to \mathcal{R}$ such that, for every $t \in \mathcal{T}$, the rules t and $f(t)$ are equal up to renaming. Such a *typed* occurrence GTS is called a *graph process* [4].

Since all graphs in the rules of Fig. 6 are well-typed over the type graph TG, their union C is typed over TG by the union of the typing homomorphisms of its subgraphs. The rules representing basic actions, like operation invocations and state transformations, can be mapped directly to rules in $\mathcal{R}$. The control flow rules, which are more complex, are mapped to compound rules derived from the elementary rules in $\mathcal{R}$.[3]

Occurrence. The occurrence of a specification-level interaction pattern at the instance level is described by a plain homomorphism of graph transformation

[3] Categorically, this typing of an occurrence GTS can be formalized as a Kleisli morphism mapping elementary rules to derived ones (see, e.g., [18,17] for similar approaches in graph transformation theory).

systems: We want the same granularity of actions on the specification and the instance level. An example of an instance-level collaboration diagram is given in Fig. 8. It does not contain additional actions (although this would be permitted

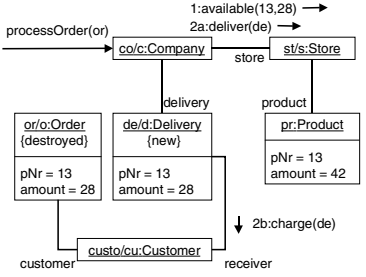


Fig. 8. Collaboration diagram on the instance level

by the definition of occurrence), but the additional context of the Product instance. In the process in Fig. 6, this would lead to a corresponding extension of the start rule.

As before, the GTS homomorphisms forming the three sides of the triangle have to be compatible. That means, an occurrence of a specification-level collaboration diagram in an instance-level one has to respect the typing of classes, associations, and attributes and of operations and basic actions.

This completes the formalization of the type-specification-instance triangle in the three views of collaboration diagrams of structure, state transformation, and interaction.

5 Conclusion

In this paper, we have proposed a semantics for collaboration diagrams based on concepts from the theory of graph transformation. We have identified and formalized three different aspects of a system model that can be expressed by collaboration diagrams (i.e., structure, state transformation, and interaction) and, orthogonally, three levels of abstraction (type, specification, and instance level). In particular, the idea of collaboration diagrams as state transformations provides new expressive power which has so far been neglected by the UML standard. The relationships between the different abstraction levels and aspects are described in terms of homomorphisms between graphs, rules, and graph transformation systems.

The next steps in this work consist in transferring the new insights to the UML specification. On the level of methodology and notation, the state transformation aspect should be discussed as one possible way of using collaboration diagrams. On the level of abstract syntax (i.e., the meta model) the pattern-occurrence relation between specification- and instance-level diagrams has to be made explicit, e.g., by additional meta associations. (In fact, this has been

partly accomplished in the most recent draft of the standard [25].) On the semantic level, a representation of the causal dependencies in a collaboration diagram is desirable which captures also the data flow between actions.

It remains to state more precisely the relation between collaboration diagrams as defined by the standard and our extended version. Obviously, although our collaboration diagrams are syntactically legal, the collaboration diagrams that are semantically meaningful according to the UML standard form a strict subset of our high-level diagrams based on graph matching. An implementation of this matching by explicit navigation (as it is given, for example, in [14] as part of a code generation in JAVA) provides a translation back to the original low-level style. The formal properties of this construction have not been investigate yet.

References

1. Action Semantics Consortium. Precise action semantics for the Unified Modeling Language, August 2000. `http://www.kc.com/as_site/`.
2. P. Bottoni, M. Koch, F. Parisi Presicce, and G. Taentzer. Consistency checking and visualization of OCL constraints. In Evans et al. [13], pages 294–308.
3. D. Coleman, P. Arnold, S. Bodof, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremes. *Object Oriented Development, The Fusion Method*. Prentice Hall, 1994.
4. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–266, 1996.
5. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation, Part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, pages 163–245. World Scientific, 1997.
6. D. D'Souza and A. Wills. *Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.
7. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*. World Scientific, 1999.
8. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), Paderborn, November 1998*, volume 1764 of *LNCS*. Springer-Verlag, 2000.
9. H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency and Distribution*. World Scientific, 1999.
10. H. Ehrig, M. Pfender, and H.J. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.
11. G. Engels, J.H. Hausmann, R. Heckel, and St. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In Evans et al. [13], pages 323–337.
12. G. Engels, R. Hücking, St. Sauer, and A. Wagner. UML collaboration diagrams and their transformation to Java. In France and Rumpe [15], pages 473–488.
13. A. Evans, S. Kent, and B. Selic, editors. *Proc. UML 2000 – Advancing the Standard*, volume 1939 of *LNCS*. Springer-Verlag, 2000.

14. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph transformation language based on UML and Java. In Ehrig et al. [8].
15. R. France and B. Rumpe, editors. *Proc. UML'99 – Beyond the Standard*, volume 1723 of *LNCS*. Springer-Verlag, 1999.
16. M. Gogolla. Graph transformations on the UML metamodel. In J. D. P. Rolim et al., editors, *Proc. ICALP Workshops 2000, Geneva, Switzerland*, pages 359–371. Carleton Scientific, 2000.
17. M. Große-Rhode, F. Parisi Presicce, and M. Simeoni. Refinement of graph transformation systems via rule expressions. In Ehrig et al. [8], pages 368–382.
18. R. Heckel, A. Corradini, H. Ehrig, and M. Löwe. Horizontal and vertical structuring of typed graph transformation systems. *Math. Struc. in Comp. Science*, 6(6):613–648, 1996.
19. R. Heckel, H. Ehrig, U. Wolter, and A. Corradini. Double-pullback transitions and coalgebraic loose semantics for graph transformation systems. *Applied Categorical Structures*, 9(1), January 2001.
20. R. Heckel and St. Sauer. Strengthening the semantics of UML collaboration diagrams. In G. Reggio, A. Knapp, B. Rumpe, B. Selic, and R. Wieringa, editors, *UML'2000 Workshop on Dynamic Behavior in UML Models: Semantic Questions*, pages 63–69. October 2000. Tech. Report no. 0006, Ludwig-Maximilians-University Munich, Germany.
21. R. Heckel and A. Zündorf. How to specify a graph transformation approach: A meta model for FUJABA. In H. Ehrig and J. Padberg, editors, *Uniform Approaches to Graphical Process Specification Techniques, satellite workshop of ETAPS 2001, Genova, Italy*, 2001. To appear.
22. A. Knapp. A formal semantics of UML interactions. In France and Rumpe [15], pages 116–130.
23. M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In *Proc. ICALP'98*, volume 1443 of *LNCS*, pages 856–867. Springer-Verlag, 1998.
24. Object Management Group. UML specification version 1.3, June 1999. `http://www.omg.org`.
25. Object Management Group. UML specification version 1.4beta R1, November 2000. `http://www.celigent.com/omg/umlrtf/`.
26. G. Övergaard. A formal approach to collaborations in the Unified Modeling Language. In France and Rumpe [15], pages 99–115.
27. V. Pratt. Modeling concurrency with partial orders. *Int. Journal. of Parallel Programming*, 15(1):33–71, February 1986.
28. W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.
29. A. Schürr, A.J. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In Ehrig et al. [7], pages 487–550.

Specification of Mixed Systems in KORRIGAN with the Support of a UML-Inspired Graphical Notation

Christine Choppy[1], Pascal Poizat[2], and Jean-Claude Royer[2]

[1] LIPN, Institut Galilée - Université Paris XIII,
Avenue Jean-Baptiste Clément, F-93430 Villetaneuse, France
Christine.Choppy@lipn.univ-paris13.fr
[2] IRIN, Université de Nantes
2 rue de la Houssinière, B.P. 92208, F-44322 Nantes cedex 3, France
{Pascal.Poizat,Jean-Claude.Royer}@irin.univ-nantes.fr
http://www.sciences.univ-nantes.fr/info/perso/permanents/poizat/

Abstract. Our KORRIGAN formalism is devoted to the structured formal specification of mixed systems through a model based on a hierarchy of *views* [4,20]. In our unifying approach, views are used to describe the different aspects of a component (both internal and external structuring). We propose a semi-formal method with guidelines for the development of mixed systems, that helps the specifier providing means to structure the system in terms of communicating subcomponents and to describe the sequential components. While there is growing interest for having both textual and graphical notations for a given formalism, we introduce composition diagrams, a UML-inspired graphical notation for KORRIGAN, associated with the various steps of our method. We shall show how our method is applied to develop a KORRIGAN specification (both in textual and graphical notation) and illustrate this approach on a case study.

Keywords: formal specification, mixed specification, graphical notation, symbolic transition systems, KORRIGAN, UML

1 Introduction

The use of formal specifications is now widely accepted in software development to provide abstract, rigorous and complete descriptions of systems. Formal specifications are also essential to prove properties, to prototype the system and to generate tests.

In the last few years, the need for a separation of concerns with reference to static (data types) and dynamic aspects (behaviours, communication) appeared. This issue was addressed in approaches combining algebraic data types with other formalisms (*e.g.* LOTOS [17] with process algebras or SDL [7] with State/Transition Diagrams), and also more recently in approaches combining Z and process algebras (*e.g.* OZ-CSP [27] or CSP-OZ [8]). This is also reflected in

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 124–139, 2001.

object oriented analysis and design approaches such as UML [28] where static and dynamic aspects are dealt with by different diagrams (class diagrams, interaction diagrams, Statecharts). However, the (formal) links and consistency between the aspects are not defined, or trivial. This limits either the possibilities of reasoning on the whole component or the expressiveness of the formalism. Some approaches encompass both aspects within a single framework (*e.g.* LTS [24], rewriting logic [19] or TLA [18]). These "homogeneous" approaches ease the verification and the definition of consistency criteria for the integration of aspects, but at the cost of a loss of expressiveness for one of the aspects or a poor level of readability. Moreover, the definition of a method remains an important lack of most of these approaches.

The KORRIGAN formalism is devoted to the structured formal specification of mixed systems through a model based on a hierarchy of *views*. Our approach aims at keeping advantage of the languages dedicated to both aspects (*i.e.* Symbolic Transition Systems for behaviours, algebraic specifications derived from these diagrams for data parts, and a simple temporal logic and axiom based glue for compositions) while providing an underlying unifying framework accompanied by an appropriate semantic model. Moreover, experience has shown that our formalism leads to expressive and abstract, yet readable specifications.

We propose, in this paper, a semi-formal method with guidelines for the development of mixed systems with KORRIGAN. This method helps the specifier providing means to structure the system in terms of communicating subcomponents and to describe the sequential components. While there is growing interest for having both textual and graphical notations for a given formalism (*e.g.* SDL and UML/XMI), we shall in the following introduce composition diagrams, a UML-inspired graphical notation for KORRIGAN, associated with the various steps of our method. We suggest to reuse some UML notation, but we also present some proper KORRIGAN graphic notations. Since our model is component based we have an approach which is rather different from UML on communications and concurrent aspects. Thus we will present specific notations to define the dynamic interface of a component, its communications with others and concurrency. We shall show how our method is applied to develop a KORRIGAN specification (both in textual and graphical notation) and illustrate this approach on a transit node case-study.

The paper is organized as follows. Section 2 presents the Transit Node case study. Section 3 gives an overview of KORRIGAN. It describes briefly the view model and details the associated specification method. Then, in Section 4 we present our UML-inspired notation diagrams for interfaces, compositions, behaviours, and communications. Finally, some related works are discussed in our conclusion.

2 The Transit Node Case Study

This case study was adapted within the VTT project from one defined in the RACE project 2039 (SPECS: Specification Environment for Communication

Software). It consists of a simple transit node where messages arrive, are routed, and leave the node (Fig. 1). We do not give the full informal specification text here but shortly describe what is needed in the context of this paper. The full presentation of the case study may be found in [20].

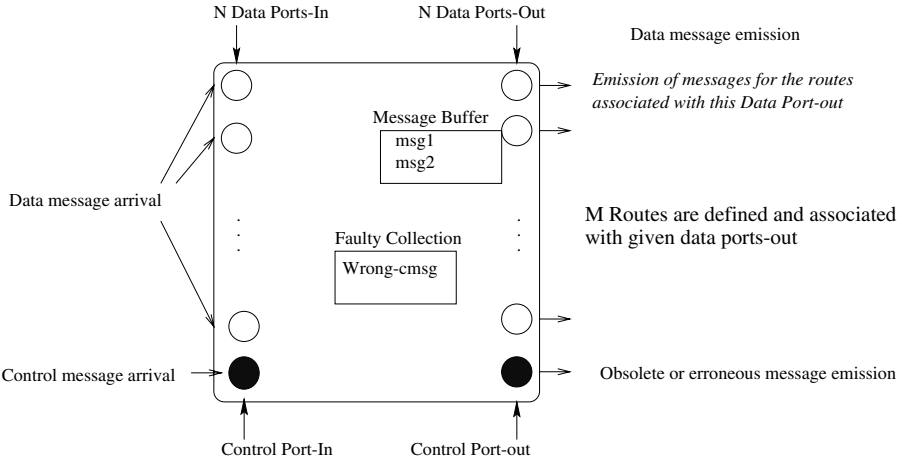


Fig. 1. Transit Node

The system to be specified consists of a transit node with one *Control Port-In* that receives control messages, one *Control Port-Out* that emits erroneous or obsolete messages, N *Data Ports-In* that receive data messages to be routed, N *Data Ports-Out* that emit data messages, and M *Routes* through. Each port is serialized, and all ports are concurrent to all others.

A data message is of the form *Route(m).Data*. The *Data Port-In* routes the message to any one of the open *Data Ports-Out* associated with the message route m where the message has to be buffered until the *Data Port-Out* can process it. When a message is erroneous (e.g. its route is not defined) or obsolete (its transit time is greater than a constant time T), it is eventually directed to a faulty collection.

The control messages modify the transit node configuration by enabling new data ports-in and out, or defining routes together with their associated data ports. The *Send-Faults* control message is used to route messages from the faulty collection to the *Control Port-Out* from which they will be eventually emitted.

3 Korrigan: A Formalism for Mixed Specification

In this Section, we will briefly present our model, the KORRIGAN specification language and the associated method.

3.1 Korrigan and the View Model

Our model [4,20] is based upon the structured specification of communicating components (with identifiers) by means of structures that we call *views* which are expressed in KORRIGAN, the associated formal language (Fig. 2).

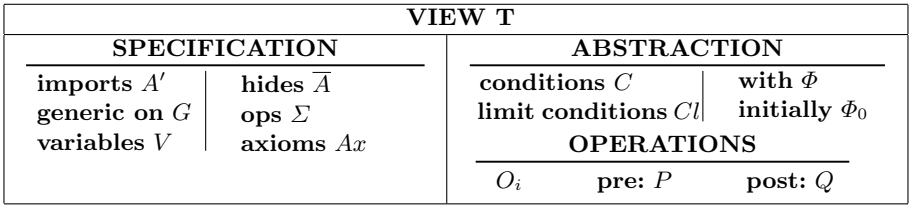


Fig. 2. KORRIGAN syntax (views)

Views use conditions to define an abstract point of view for components. These conditions are also used to define an inheritance relation for views. STS, *i.e. Symbolic Transition Systems*[1] are built using the conditions [21]. The main interest with these transition systems is that (i) they avoid state explosion problems, and (ii) they define equivalence classes (one per state) and hence strongly relate the dynamic and the static (algebraic) representation of a data type.

Views are used to describe in a structured and unifying way the different aspects of a component using "internal" and "external" structuring. We define an *Internal Structuring View* abstraction that expresses the fact that, in order to design a component, it is useful to be able to express it under its different aspects (here the static and dynamic aspects, with no exclusion of further aspects that may be identified later on). Another structuring level is achieved through the *External Structuring View* abstraction, expressing that a component may be composed of several subcomponents. Such a component may be either a global component (integrating different internal structuring views in an *Integration View*), or a composite component (*Composition View*). Integration views follow an *encapsulation principle*: the static aspect (*Static View*) may only be accessed through the dynamic aspect (*Dynamic View*) and its identifier (`Id`). The whole class diagram for the view model is given in Figure 3.

Components are "glued" altogether in external structuring views (Fig. 4) using both axioms and temporal logic formulas. This glue expresses a generalized form of synchronous product (for STS) and may be used to denote different concurrency modes and communication semantics. The δ component may be either `LOOSE`, `ALONE` or `KEEP` and is used in the operational semantics to express different concurrency modes (synchronous or asynchronous modes) and communication schemes. The `axioms` clause is used to link abstract guards that may

[1] Mainly transition systems with guarded transitions and open terms in states and transitions, see Figure 12 or [4].

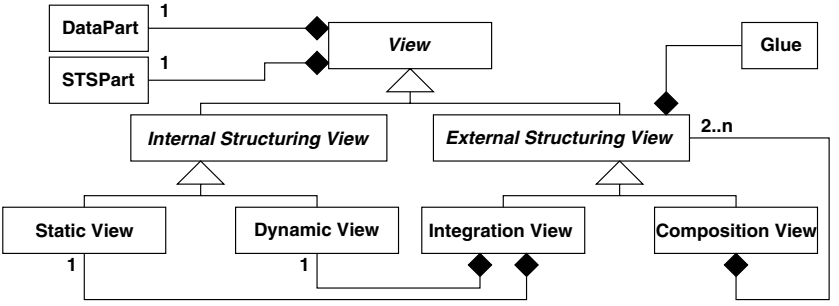


Fig. 3. Views class hierarchy (UML notation)

exist in components with operations defined in other components. The Φ and Φ_0 elements are state formulas expressing correct combinations of the components conditions (Φ) and initial ones (Φ_0). The Ψ element is a set of couples of transition formulas expressing what transitions have to be triggered at the same time (this expresses communication). The `COMPOSITION` clauses may use a syntactic sugar: the *range* operator (`i:[1..N]` or `i:[`e_1`, ... ,`e_n`]`), a bounded universal quantifier.

EXTERNAL STRUCTURING VIEW T	
SPECIFICATION	**COMPOSITION** δ
imports A' | **variables** V **generic on** G | **hides** $\overline{A}$	**is** $id_i : Obj_i\langle I_i\rangle$ | **axioms** Ax_Θ **with** Φ, Ψ **initially** Φ_0

Fig. 4. KORRIGAN syntax (compositions)

3.2 A Method for the Specification of Mixed Components

Methods are needed to help using formal specifications in a practical way. We propose a method for the development of mixed systems, that helps the specifier providing means to structure the system in terms of communicating subcomponents and to give the sequential components using a semi-automatic concurrent automata generation with associated algebraic data types. A previous version of our method [21] is described in terms of the agenda[2] concept [12,14]. The method presented here is refined and accompanied with the use of visual diagrams.

Our method mixes constraint-oriented, resource-oriented and state-oriented specification styles [29,30] and produces a modular description with a dynamic behaviour and its associated data type. Our method is composed of four steps (Fig. 5), with associated diagrams, for obtaining the specification.

[2] Agendas describe a list of activities for solving a task in software engineering, and are developed to provide guidance and support for the application of formal specification techniques.

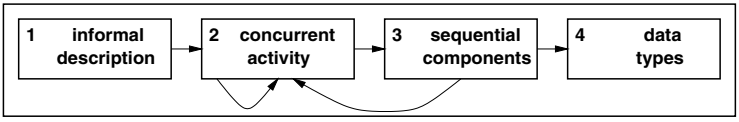


Fig. 5. Method Step Dependencies at the Overall Level

These steps correspond to:

- the **informal description** of the system to be specified. The aim of this first step is to sketch out the system characteristics (data, constraints and functionalities).
- the **concurrent activity** description. The idea is to define the *system architecture* in terms of a decomposition tree, and then to add *communication information* between the elements of this tree. This part on communication may be achieved after some basic components have been described (or reused). This step is composed of three sub-steps that may be iterated (Fig. 6).

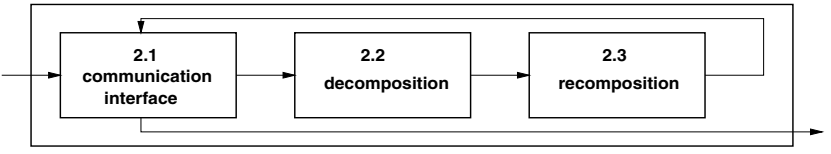


Fig. 6. Method Step Dependencies for the Concurrent Activity

In the first one (2.1), the *communication interface* of the component being described is given in term of an *interface diagram*.
Then, in a second step (2.2), the component is decomposed into concurrent (possibly communicating) subcomponents using *composition diagrams*. Once a sequential component is obtained, a further decomposition step is applied to reflect the fact that components are indeed the integration of different aspects (static and dynamic). Therefore, we have *concurrent decompositions* (described by *concurrent composition diagrams*) and *integration decompositions* (described by *integration composition diagrams*), *i.e.* separation of aspects steps.
In the third step (2.3), the different composition diagrams (either integration or concurrent composition diagrams) are completed with the communications that take place between them. This can be done by reusing communication patterns. When they are completed with communication information, the composition diagrams are called *communication diagrams*. Note that in our KORRIGAN framework, both concurrency and integration are specified in a unified way.

Table 1 gives the correspondence between the concurrent activity sub-steps, the corresponding diagrams, and the corresponding KORRIGAN view structures. All the diagrams will be presented in the next Section.

Table 1. Method steps, diagrams and KORRIGAN view structures

step	diagram	Korrigan
communication interface	interface (e.g. Fig. 8)	Σ in Internal Structuring Views `SPECIFICATION` part
decomposition	composition (e.g. Fig. 9)	External Structuring View (partial: `imports` and `is` clauses)
recomposition	communication (e.g. Fig. 14)	External Structuring View (full)

- the **sequential component** descriptions. The term "condition" refers to preconditions required for a communication to take place, and also to conditions that affect the behaviour when a communication takes place. The operations are defined in terms of pre and postconditions over these conditions. These concepts correspond to our formal language (see Fig. 2) but gives a general method for a wider set of mixed specification languages (it has been applied to LOTOS and SDL). A guarded automaton is then progressively and rigorously built from the conditions. Type information and operation preconditions are used to define the automaton states and transitions. A dynamic behaviour (described by a *behavioural diagram*) may then be computed from the automaton using some standard patterns.
- the **data type** (functional) specifications. The last improvement is the assisted computation of the functional parts (in KORRIGAN, the data type parts of views). Our method reuses a technique [2] which allows one to get an abstract data type from an automaton. This technique extracts a signature and generators from the automaton. Furthermore, the automaton drives the axiom writing so that the specifier has only to provide the axioms right hand sides.

After a preliminary presentation of our UML-inspired notation, we introduce the various diagrams supporting our formal specification method illustrated with the Transit Node case study.

4 A UML-Inspired Graphical Notation

UML [28] is a notation to be used for object-oriented analysis and design. Since it is very expressive (it has 11 different diagram types), and its (informal) semantics is unclear in some cases [23], it is common to restrict oneself to a subset of it, but there are also proposals to modify/extend it [11,16].
We think that, in complement to the theoretical approach that tries to formalize

the UML [22,9], an interesting and more pragmatic approach is to reuse existing well-accepted semi-formal notations and use them as a graphical means to improve the readability of formal languages and concepts. In order to be close to the UML, we select a subset of UML that is relevant to illustrate concepts of our approach. We also extend/modify it when needed. For instance (Fig. 8), we use a simple class diagram together with informations related to the possible communications (its interface) of the component. The interface symbols [3] we use are described in Figure 7.

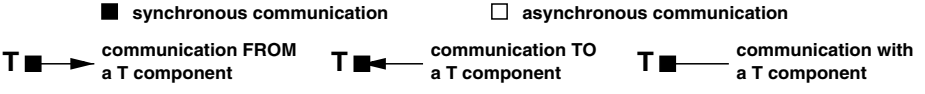


Fig. 7. Communication Interfaces

There exist different kinds of compositions within the set-theoretic or the object-oriented framework. We restrict ourselves to the strong composition of critical systems and distributed applications, that is composition with dependence, exclusivity and predominance. This can be compared with the strong composition in UML (black diamond), therefore, we use this UML symbol to represent it (Fig. 9). However, our approach is more component-oriented than UML since we explicitly address communication issues in interfaces (Fig. 8) and concurrency in communication diagrams (Fig. 15) whereas in UML they are embedded within Statecharts.

4.1 Interface and Composition Diagrams

We introduce composition diagrams to decompose a component into subcomponents. We denote by (de)composition both the separation/integration of the different aspects of a component (static and dynamic), and the (de)composition into concurrent communicating subcomponents. The KORRIGAN model enables us to describe both, in a unifying way, using specific External Structuring Views, respectively Integration and Composition Views.

Interface Diagrams. Following our method, at an abstract level of description, the transit node may be described using its data and its functionalities.

We may now give the description of the transit node at the most abstract level (Fig. 8). It has four functionalities. Its data are composed of three lists. The transit node is parameterized by `N`, the maximal number of ports within.

[3] Note that KORRIGAN has no support for asynchronous communication, hence it is achieved through buffers.

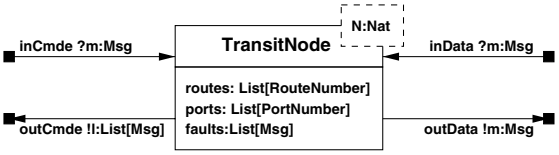


Fig. 8. `TransitNode` Interface Diagram

Composition Diagrams. Following our method, we decompose the transit node into control ports and data ports using two views: `ControlPorts` and `DataPorts`. We then distribute the transit node functionalities and data in them. Finally, we name the subcomponents of the transit node (`control` and `data`). Figure 9 represents the first level of decomposition of the transit node.

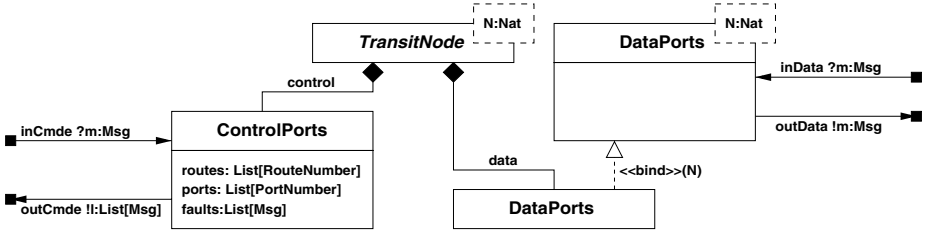


Fig. 9. `TransitNode` Composition Diagram

At this step, we may retrieve a partial KORRIGAN specification for the transit node using its graphical representation. Such a specification would be partial because the component views it uses (`imports` clause) may have not been defined yet. The communications between the subcomponents (that is the "glue" between them) will be specified in a latter step but this does not always implies that the super-component is partial because there may not be any communications at all between two subcomponents.

Applying the same decomposition process on the `DataPorts` view, we obtain the Figure 10 diagram. Note here that its subcomponents make use of the range operator to express a set of identifiers. The `OutputDataPort` has a buffer to deal with its serialization constraints. We do not treat the `ControlPorts` here by lack of place, see [20].

Integration Composition Diagrams. As mentioned above, in KORRIGAN the integration of the different aspects of components within a global one is also a kind of composition. However, to distinguish between integration and concurrent composition, we use integration diagrams: composition diagrams where the names of the integration components are put into gray boxes.

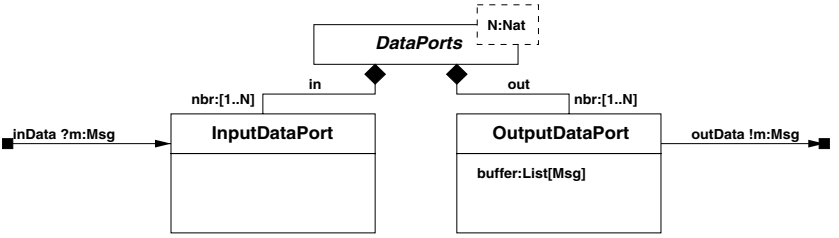


Fig. 10. `DataPorts` Composition Diagram

In the integration diagrams, the data that were in previous composition diagrams are transformed into corresponding static aspects. It is a matter of design to choose if each data will have its corresponding static aspect view, or if several may be incorporated into a single static aspect view. This last solution complicates the description of the communication between aspects and diminishes the reusability level of the components. It is important to note that when a component has no data, it is integrated with a trivial (`Null`) static part. See [20] for the case study diagrams.

4.2 Behavioural Diagrams

Basic components are specified using views. Such a view may be given as a triple (`SPECIFICATION`, `ABSTRACTION` and `OPERATIONS` parts), or using the STS derivation principles [21], as a couple (`SPECIFICATION` part and a STS). These STSs may be related to Statecharts ([13], or UML ones) but for some differences:

- STSs are simpler (but less expressive) than Statecharts;
- STSs model sequential components (concurrency is done through external structuring and the computation of a structured STS from subcomponents STSs [4]);
- STSs are built using conditions which enable one to semi-automatically derive them from requirements;
- STSs may be seen as a graphical representation of an abstract interpretation of an algebraic data type [2].

We give in Figure 11 a part of the `InputDataPort` KORRIGAN specification, and in Figure 12 the corresponding behavioural diagram.

In presence of generic components, we may use instantiation diagrams to relate concrete components to their generic parent. Such views are reusable. Generally, the static views are sets, lists or buffers, *i.e.* the description in dynamic terms of the inputs and outputs of a storage element. In Figure 13, the instantiation process is used for the different static views describing lists.

There are also inheritance diagrams in our model [20].

DYNAMIC VIEW `InputDataPort`

SPECIFICATION

imports
Msg, RouteNumber, PortNumber
variables fc: FaultyCollection
ops
enable
FROM InputControlPort
inData ?m:Msg
FROM InputControlPort
askRoute !r:RouteNumber
TO InputControlPort
replyRoute ?l:List[PortNumber]
FROM InputControlPort
wrongRoute !m:Msg
TO FaultyCollection
correct !m:Msg
TO OutputDataPort
axioms see [20]

ABSTRACTION

conditions
enable, received, asked, replied, routeErr
with
replied ⇒ enabled
asked ⇒ received
replied ⇒ asked
routeErr ⇒ replied
initially ¬ enabled

OPERATIONS

enable
pre: true
post: enable' : true
inData
pre: enable ∧ ¬ received
post: received' : true
...

Fig. 11. `InputDataPort` in KORRIGAN

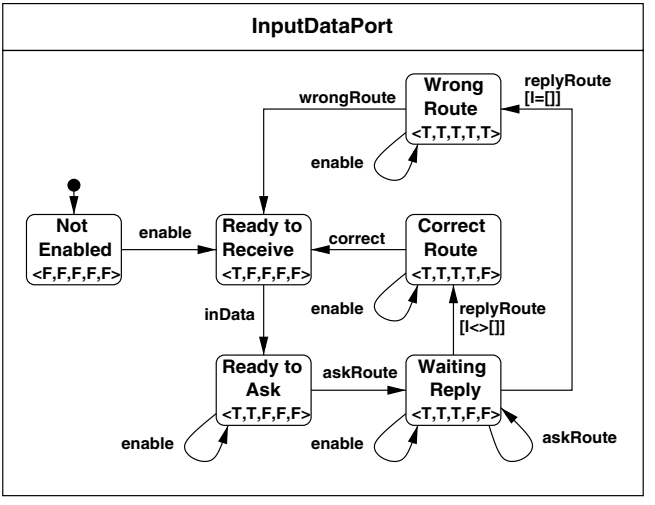


Fig. 12. `InputDataPort` Behavioural Diagram (STS)

4.3 Communication Diagrams

The communication diagrams are used to complement the composition diagrams with the inter-component communication and concurrency schemes. They use a graphical notation of the KORRIGAN glue rules (`COMPOSITION` parts in external structuring views, Fig. 4).

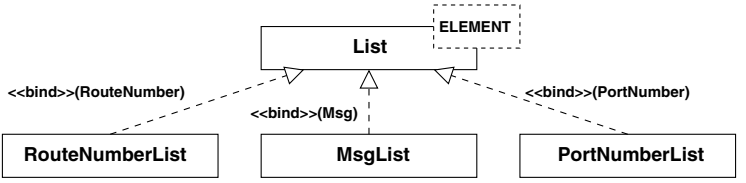


Fig. 13. `List` Instantiation Diagram

The axiomatic part of the glue (the `axioms` clause), the state temporal formulas (Φ and Φ_0), and the concurrency mode (δ) are put in the aggregating component (*i.e.* `DataPorts` for the `InputDataPort` and `OutputDataPort` views) as shown in Figure 14. Here there are no glue axioms.

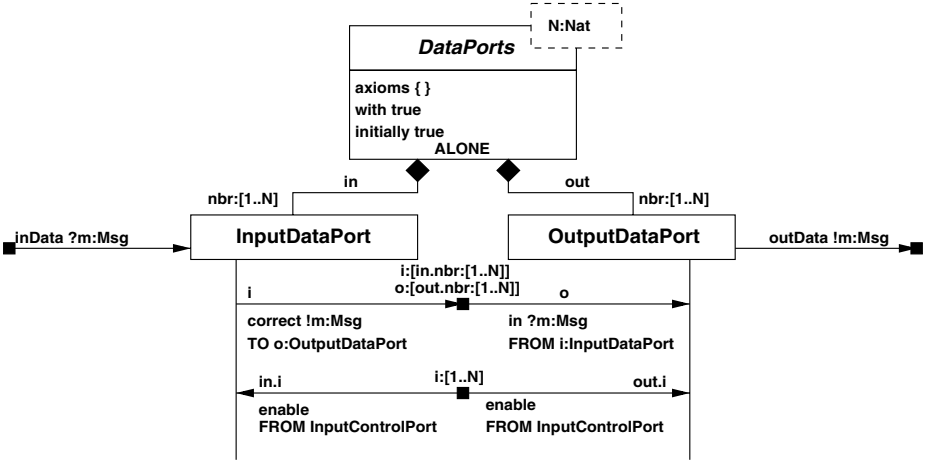


Fig. 14. `DataPorts` Communication Diagram

Each element of the transition couples (Ψ) is treated by linking the involved components with a node (boxes in Fig. 7). The parts of the couple relative to each of these components is put on the lines. Only the links between input and output ports have been represented, for example the communication between an input data port routing a correct message to the corresponding ouput data port. The elements above the links represent the participating components. Here we use again range operators as a syntactical shorthand (one link is used in place of the `N`×`N` that would be used without it):

$$\forall nbr_i \in [1..N],\ \forall nbr_o \in [1..N],\ \forall m : Msg\ .\ i = in.nbr_i,\ o = out.nbr_o\ |$$
$$\text{i.correct !m : Msg TO o} \longrightarrow\!\blacksquare\!\longrightarrow \text{o.in ?m : Msg FROM i}$$

When components at different levels are involved, for example to treat the communication between an input control port enabling[4] both a given input data port and a given output data port, we adopt a structured communication scheme and do not add more links on the communication nodes. We add the communication information on parents (in the decomposition tree) of the concerned subcomponents, here `TransitNode` (Fig. 15). The obtaining of the KORRIGAN specification for `DataPort` from its diagram is straightforward (Fig. 16).

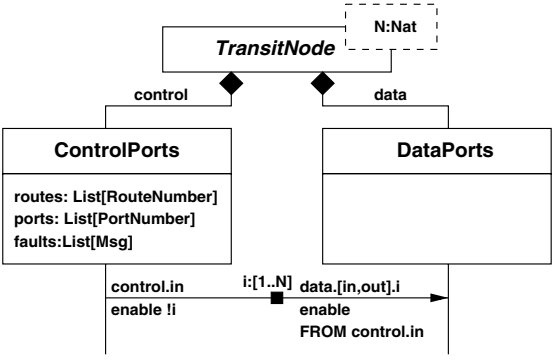


Fig. 15. `TransitNode` Communication Diagram (partial)

COMPOSITION VIEW DataPorts	
SPECIFICATION	**COMPOSITION ALONE**
imports InputDataPort, OutputDataPort **variables** N : Natural	**is** in.nbr[1..N] : InputDataPort out.nbr[1..N] : OutputDataPort **with** true,{ i:[1..N].(in.i.enable **from** s:Server, out.i.enable **from** s:Server), (i:[1..N].o:[1..N].(i.correct !m **to** o:OutputDataPort, o.correct ?m **from** i:InputDataPort), ...}

Fig. 16. `DataPorts` in KORRIGAN

Since KORRIGAN treats in a unified way both integration composition and concurrent composition, then the process and notations we have presented on

[4] This enabling scheme is used to implement the creation of object since there is no direct support for this in KORRIGAN.

concurrent communication apply on integrations too (*i.e.* for example, we would have communication diagrams between an `OutputDataPort` and its `MsgList`).

This representation of the communication is expressive enough to describe different kinds of communication, for example, both point-to-point communication (`ptp`) and broadcast communication (`broadcast`) in a client-server pattern [20]. Such a pattern may be used in the recomposition step of our method.

5 Conclusions and Related Work

We defined in previous works a formal approach based on view structures for the specification of mixed systems with both control, communications and data types. The corresponding formal language, KORRIGAN, allows one to describe systems in a structured and unifying way.

In order to make formal methods more used in the industrial world, we agree with [3]: *most important properties of specifications methods are not only the underlying theoretical concepts but more pragmatics issues such as readability, tractability, support for structuring, possibilities of visual aids and machine support*. Therefore, we have built a software environment, ASK [5], for the development of our KORRIGAN specifications.

In this paper, we propose a semi-formal method with guidelines for the development of mixed systems with KORRIGAN. Our method is supported by a UML-inspired graphical notation. We suggest, when possible, to reuse the UML notation, but we also present some proper extensions. Since our model is component-based, we have an approach which is rather different from UML on communications and concurrent aspects. Thus we also describe specific notations to define dynamic interfaces of components, communications patterns and concurrency. Our method is here applied to develop both textual and graphical specifications and illustrated on a transit node case-study.

Our concerns about methods and graphical notations for formal languages are close to [24,6] ones. However, we think we can reuse UML notations, or partly extend it using stereotypes, rather than defining new notations. Moreover, our approach is complementary to the theoretical approaches that try to formalize the UML. Our notations are also more expressive and abstract than [24] as far as communication issues are concerned.

KORRIGAN and UML-RT [25] partly address the same issues : architectural design, dynamic components and reusability. However, UML-RT is at the design level whereas KORRIGAN is rather concerned about (formal) specification issues. There are also some other difference, mainly at the communication level, but the major one is that, to the contrary of UML-RT, KORRIGAN provides a uniform way to specify both datatypes and behaviours.

Our notation for the glue between communicating components may be also related to [10]. The main differences are that our glue is more expressive than LOTOS synchronizations, and that we have a more structured organization of communication patterns.

We are now working on validation and verification procedures for our KORRIGAN specifications. Due to the use of STS, *i.e.* Symbolic Transition Systems, such procedures have to be adapted [15,26]. We also investigate the automatic translation of KORRIGAN specifications into PVS following the [1] methodology.

References

1. Michel Allemand. Verification of properties involving logical and physical timing features. In *Génie Logiciel & Ingénierie de Systèmes & leurs Applications, ICSSEA'2000*, 2000.
2. Pascal André and Jean-Claude Royer. A First Algebraic Approach to Heterogeneous Software Systems. 14th International Workshop on Algebraic Development Techniques (WADT'99), Bonas, France, 1999.
3. M. Broy. Specification and top down design of distributed systems. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *TAPSOFT'85*, volume 185 of *Lecture Notes in Computer Science*, pages 4–28. Springer-Verlag, 1985.
4. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. A Global Semantics for Views. In T. Rus, editor, *International Conference on Algebraic Methodology And Software Technology (AMAST'2000)*, volume 1816 of *Lecture Notes in Computer Science*, pages 165–180. Springer-Verlag, 2000.
5. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. The KORRIGAN Environment. *Journal of Universal Computer Science*, 2001. Special issue: Tools for System Design and Verification, ISSN: 0948-6968. to appear.
6. Eva Coscia and Gianna Reggio. JTN: A Java-Targeted Graphic Formal Notation for Reactive and Concurrent Systems. In Jean-Pierre Finance, editor, *Fundamental Approaches to Software Engineering (FASE'99)*, volume 1577 of *Lecture Notes in Computer Science*, pages 77–97. Springer-Verlag, 1999.
7. Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL : Formal Object-oriented Language for Communicating Systems*. Prentice-Hall, 1997.
8. C. Fischer. CSP-OZ: a combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 423–438, Canterbury, UK, 1997. Chapman & Hall.
9. R. France and B. Rumpe, editors. *UML'99 – The Unified Modelling Language*, volume 1723 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
10. Hubert Garavel and Mihaela Sighireanu. A Graphical Parallel Composition Operator for Process Algebras. In *Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification (FORTE/PSTV'99)*, 1999.
11. Gianna Reggio and Egidio Astesiano. An extension of UML for modelling the non purely reactive behaviour of active objects. Semi-Formal and Formal Specification Techniques for Software Systems, Dagstuhl Seminar 00411, Report No. 288, H. Ehrig, G. Engels, F. Orejas, M. Wirsing, October 2000.
12. Wolfgang Grieskamp, Maritta Heisel, and Heiko Dörr. Specifying Embedded Systems with Statecharts and Z: An Agenda for Cyclic Software Components. In Egidio Astesiano, editor, *FASE'98*, volume 1382 of *Lecture Notes in Computer Science*, pages 88–106. Springer-Verlag, 1998.
13. David Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

14. Maritta Heisel. Agendas – A Concept to Guide Software Development Activities. In R. N. Horspool, editor, *Systems Implementation 2000*, pages 19–32. Chapman & Hall, 1998.
15. M. Hennessy and H. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.
16. Bogumila Hnatkowska and Huzar Zbigniew. Extending the UML with a Multicast Synchronisation. In T. Clark, editor, *Proceedings of the third Rigorous Object-Oriented Methods Workshop (ROOM)*, BCS eWics, ISBN: 1-902505-38-7, 2000.
17. ISO/IEC. LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807, International Organization for Standardization, 1989.
18. Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.
19. José Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *CONCUR'96 : Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372, Pisa, Italy, 1996.
20. Pascal Poizat. KORRIGAN*: a Formalism and a Method for the Structured Formal Specification of Mixed Systems*. PhD thesis, Institut de Recherche en Informatique de Nantes, Université de Nantes, December 2000. in French.
21. Pascal Poizat, Christine Choppy, and Jean-Claude Royer. From Informal Requirements to COOP: a Concurrent Automata Approach. In J.M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems*, volume 1709 of *Lecture Notes in Computer Science*, pages 939–962, Toulouse, France, 1999. Springer-Verlag.
22. G. Reggio, M. Cerioli, and E. Astesiano. An Algebraic Semantics of UML Supporting its Multiview Approach. In D. Heylen, A. Nijholt, and G. Scollo, editors, *Twente Workshop on Language Technology, AMiLP 2000*, 2000.
23. G. Reggio and R. Wieringa. Thirty one Problems in the Semantics of UML 1.3 Dynamics. In *OOPSLA'99 workshop "Rigorous Modelling and Analysis of the UML: Challenges and Limitations"*, 1999.
24. Gianna Reggio and Mauro Larosa. A graphic notation for formal specifications of dynamic systems. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Formal Methods Europe (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*, pages 40–61. Springer-Verlag, 1997.
25. Bran Selic and Jim Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Technical report, Rational Software Corp., 1998.
26. Carron Shankland, Muffy Thomas, and Ed Brinksma. Symbolic Bisimulation for Full LOTOS. In *Algebraic Methodology and Software Technology (AMAST'97)*, volume 1349 of *Lecture Notes in Computer Science*, pages 479–493. Springer-Verlag, 1997.
27. Graeme Smith. A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Formal Methods Europe (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*, pages 62–81. Springer-Verlag, 1997.
28. Rational Software. Unified Modeling Language, Version 1.3. Technical report, Rational Software Corp., http://www.rational.com/uml, June 1999.
29. Kenneth J. Turner, editor. *Using Formal Description Techniques, An introduction to Estelle, LOTOS and SDL*. Wiley, 1993.
30. C. A. Vissers, G. Scollo, M. Van Sinderen, and E. Brinksma. Specification Styles in Distributed Systems Design and Verification. *Theoretical Computer Science*, 89(1):179–206, 1991.

On Use Cases and Their Relationships in the Unified Modelling Language

Perdita Stevens*

Division of Informatics
University of Edinburgh

Abstract. In the Unified Modelling Language, use cases are provided as a way of summarising the requirements on a system. They are defined informally as specifications of sets of sequences of actions, and several relationships between use cases are informally defined. Dependable, tool-supported software development necessitates precise definitions of all these concepts but the topic has so far received little attention in the literature, beyond what is present in Catalysis. This paper explores how these notions can be formalised whilst staying as close as possible to the UML standard, makes some suggestions and raises further questions.

1 Introduction

The Unified Modelling Language has been widely adopted as a standard language for modelling the design of (software) systems. Nevertheless, certain aspects of UML are not yet defined precisely. This paper is concerned with one such aspect: use cases and their relationships. The relationship between a use case and its constituent actions, and especially the implications of this for the relationships between use cases, are a particularly frequent source of confusion; we have not reached the point where a tool could support the use of use cases without making major decisions concerning how to interpret the UML standard.

In this paper we discuss how use cases and their relationships may be formalised. We aim to use the minimum of formal machinery possible; so, for example, we model use cases as plain labelled transition systems, rather than as processes in some particular process algebra. [1] A major aim of this work is to be faithful to the UML standard wherever possible: UML is a hard-fought compromise and it seems sensible to try to formalise what is currently intended rather than, or at least before, suggesting improvements to that intention. There are, however, several points at which our formalisation attempt forces us to conclude that the UML standard needs amendment.

* Email: Perdita.Stevens@dcs.ed.ac.uk. Fax: +44 131 667 7209

[1] Given that process algebras are typically explained and given semantics using labelled transition systems, working directly with the LTSs is indeed "more basic", requiring strictly less machinery: it gives us a well-defined notion of what it would mean for an alternative translation into a process algebra to be consistent with our decisions, without requiring us to enter into the process algebra wars.

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 140–155, 2001.

The paper is structured as follows. The remainder of this section includes a note on standards and terminology, and a brief discussion of related work. In Sect. 2, we discuss the way UML sees use cases as made up of constituent actions. We model use cases and raise questions concerning their relationship with the system that provides them. In Sect. 3, we go on to discuss the "dependency" relationships between use cases, covering ≪include≫ in detail. In Sect. 4 we consider the more controversial relationship, generalisation between use cases; here we fail to find an adequately simple formalisation of UML's intentions, and indeed expose some problems with even the informal description. Finally in Sect. 5 we conclude and discuss future work.

1.1 Note on Standards and Terminology

This paper is based on UML1.3, the current standard at the time of writing. It is hoped that this work may feed into the ongoing construction of UML2.0, a major aim of which is to increase the precision of UML. The reader is assumed to be familiar with UML and should, in particular, be aware that UML's treatment of use cases changed significantly between versions 1.1 and 1.3, in response to criticisms of UML1.1's treatment: many UML books, however, still refer to UML1.1. We will refer to [10] as evidence of the intentions of some of the authors of UML, but it is important to note that the definition of UML is the OMG standard [7], not what is contained in any UML book. We cite material from [7] by page; thus, [7] (3-90) indicates page 90 of Sect. 3 of the UML standard, that is, of the Notation Guide.

Catalysis is described principally in [5]. Readers need to be aware that the notation used there, though UML-like, is not standard UML; and also that some technical terms have different meanings in UML and in Catalysis. Unless otherwise stated, this paper uses the UML notions.

1.2 Related Work

Perhaps surprisingly, given the frequency of mailing list questions and informal discussions of use cases and their relationships, there seems to be no previous work that formalises use cases in a way comparable to the present. In practice, many use case experts such as Alistair Cockburn (see e.g. [4]) advocate an informal approach to use cases, in which use cases are simply considered as tasks. This is arguably the right approach for a practitioner to take; but even so, see Sect. 5 for Cockburn's comment on deficiencies in current understanding of generalisation of use cases. The closest comparator to the present work is Gunnar Övergaard's recent PhD thesis [9], which reached the author too late to be considered in detail here. At first sight, a major difference is that Övergaard's formalisation does not seem to consider fully the implications of the dynamic choices that are made by the parties to a use case. The thesis supercedes an earlier paper [8], which formalised some aspects of use cases in UML1.1 and was influential on the development of UML1.3, considered here. The authors of the development

method Catalysis [5], have also been more concerned with preciseness than many of those involved in UML: we will naturally turn to Catalysis for ideas.

There has been more work that formalises UML activity diagrams. We mention particularly Bolton and Davies' work defining activity diagrams as CSP processes [3]. In the present work we are interested, as stated, in formalising use cases using the minimal machinery possible, so we refrain from introducing any particular process algebra for describing use cases; however, it would be interesting in future to combine the two approaches, especially with the aim of formalising the interactions between different use cases in an overall system workflow. Note that in the present work we do not consider interactions or interference between use case instances, either of the same or different use cases, at all.

Theoretical work which is relevant to particular sections of this work is discussed in those sections.

2 What Is a Use Case and What Is It Made of?

A use case diagram gives a high-level view of the requirements of a system: it shows the *actors* of a system – that is, the roles played by humans or external systems that interact with it – and the tasks, or use cases, which the instances of the system and actors collaborate to perform. Not all actors need be involved in all use cases; an association between an actor and a use case shows the potential for communication between instances of the use case and the actor. Use cases may in fact be used to describe the requirements on subsystems, components etc. (that is, on any Classifier), so we follow the UML standard and use the term "entity" to describe the system, subsystem, component etc. whose requirements are documented by a use case.

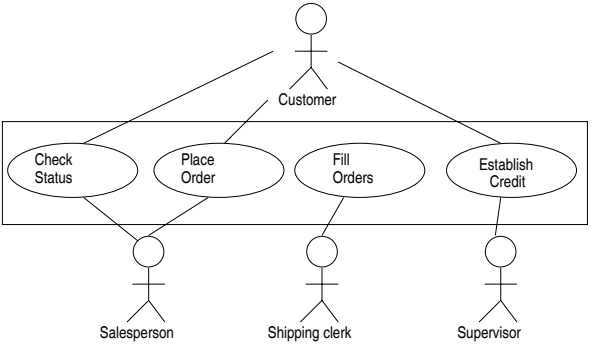


Fig. 1. Basic use case diagram, adapted from [7] (3-90)

The most crucial passages on use cases in the standard are the following:

> The use case construct is used to define the behavior of a system or other semantic entity without revealing the entity's internal structure. Each use case specifies a sequence of actions, including variants, that the entity can perform, interacting with actors of the entity. In the metamodel UseCase is a subclass of Classifier, specifying the sequences of actions performed by an instance of the UseCase. The actions include changes of the state and communications with the environment of the UseCase.

[7] (2-120)

> Each use case specifies a service the entity provides to its users, i.e. a specific way of using the entity. The service, which is initiated by a user, is a complete sequence. This implies that after its performance the entity will in general be in a state in which the sequence can be initiated again. A use case describes the interactions between the users and the entity as well as the responses performed by the entity, as these responses are perceived from the outside of the entity. A use case also includes possible variants of this sequence, e.g. alternative sequences, exceptional behavior, error handling etc.

[7] (2-124)

Like most design notations, use cases are used in two different ways: (a) by someone wanting to *design* the entity, to describe what a correct design must do; or (b) by someone wanting to *use* the entity as a black box – for example, in the design of another entity – to describe the services offered and their correct use. The main difference between the needs of these users will be the nature of the actions involved: the designer will probably begin with abstract, conceptual actions which may never have an exact counterpart in any implementation (even the direction of a communication may not have been decided when the first use cases are described [5]) whereas the user will require a description in which the actions do correspond directly to communications with the entity. The distinction will not be important for our purposes here, however, where action can be taken in either sense.

It is also said ([7] 2-120,2-125) that use cases may be described in a variety of ways – for example, text, state charts, pre- and post-conditions – which may shed some light on what a use case is intended to be, as we shall discuss. None of these descriptions is privileged as a definition, however.

The first thing to note is that a use case has a dual existence, like other Classifiers (classes, components, etc). On one hand, it may often usefully be identified with the set of its instances. That is, just as it is sometimes useful to identify a class with the set of objects in that class, it is sometimes useful to identify a use case with the set of instances of the use case. An instance of UseCase is one particular (maximal) sequence of actions from the many which comprise the use case. More precisely a UseCaseInstance is

> the performance of a sequence of actions specified in a use case. [...] An explicitly described UseCaseInstance is called a scenario.

(2-120) One may think of a UseCaseInstance as a single-branch labelled transition system, and of a scenario as the sequence of labels on the branch; the difference will not be important for our purposes.

On the other hand, just as classes may often be more usefully regarded as "things" in their own right – collections of attribute and operation specifications and implementations – use cases too may be studied as Classifiers, neglecting their instances.

We will consider the two aspects in more detail and propose a synthesis.

2.1 Use Cases as Classifiers

In the UML metamodel, UseCase is a specialisation of Classifier. This implies, among other things, that use cases can have structural and behavioural features, such as attributes and operations.

Structural features. The attributes of a use case are the means by which it is recorded how the state of a use case changes as a sequence of actions is carried out. In practice, it is usual to regard the entity and the actors as having certain conceptual attributes – conceptual in that there is no intention that a given attribute has a direct counterpart in any implementation – and then the state space of one of the use cases of the entity should induce an equivalence relation on the set of (joint) conceptual states of the entity and the actor instances. Notice, here, the subtlety introduced by the lack of any constraint in UML on the multiplicity of associations between actors and use cases: the number of instances of a given actor (potentially) communicating with a given instance of a use case might even be unbounded! In practice it is hard to imagine an example where more than one *instance of* a given actor communicates with a given *instance of* a use case (there are none in [7] or [10] for example). If there are two distinct such actor instances then almost by definition they play different roles in the use case and should therefore be modelled as instances of different actors. We suggest that this should be enforced in UML by constraining the actor end of any association between a use case and an actor to have multiplicity either "0,1" or "1", and will assume this constraint.

Behavioural features. In the UML metamodel an operation is an example of a behavioural feature. We omit to explain the details here: suffice it to say that the behavioural features of a use case will specify those actions that the entity can send and receive which are relevant to the use case. We will work simply with unanalysed actions; as with states, different choices of high-level syntax are possible to reflect the structure of actions as, for example, operation invocations with particular parameter values. Notice that there is in particular no general expectation that an action should have a reply, though a high level use case specification language that translated down into our basic formalisation might impose such restrictions when appropriate.

2.2 Use Cases as Specifications of Sequences of Actions

A use case "specifies a sequence of actions with variants". Taken at face value, this suggests that what is needed is a definitive way to decide, given a use case and a sequence of actions, whether the sequence of actions matches the specification given by the use case.

However, we must take more care over "with variants". In practice, use cases are normally described in English or pseudocode and include statements like "if [something happens within a calculation] the system ..." and "the user enters a datum and the system acts accordingly: ...". That is, the points at which the variations may happen (the branch points) are specified.

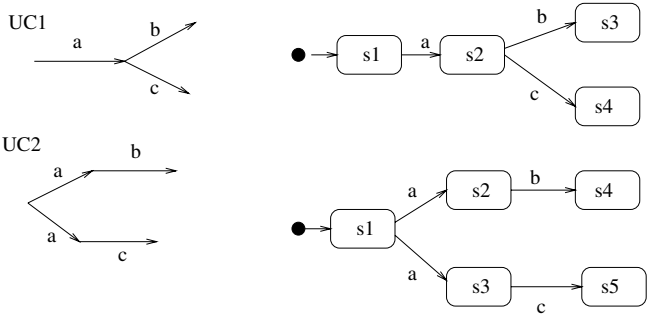


Fig. 2. Location of branch points

To illustrate the implications of this, let us represent the choices as branch points in a labelled transition system, or equivalently as states with more than one output transition in a statechart. (We illustrate both this one time to show that it is possible to use existing UML notions: however, we do not recommend the use of UML statecharts for this purpose, because their extra power relative to LTSs is not required and because their correspondingly complex semantics makes misunderstandings more likely.) The top line of Fig. 2 specifies the same set of sequences of actions – $\{ab, ac\}$ – as the bottom line. However, we argue that they should not be regarded as interchangeable. Suppose that each of $UC1$ and $UC2$ are representations of a use case offered by a system, S, and that b and c represent "input" actions chosen by (an instance of) actor A (for the sake of argument, and to make it clear that it is not reasonable to expect A to "react intelligently", let A be an external system with which S must communicate). Then if what S provides is $UC1$, A does not have to commit to entering b or c until after the action a has happened; the choice between b and c may be made, for example, after some computation within the actor that involves third parties outside the scope of a model of S and does not begin until after a has occurred (and this might be essential: a could be an "output" action which carries information that A uses to choose the third party). However, an implementation of A that behaves this way may fail if it tries to carry out $UC2$: by the time the

action a has happened, a commitment may have been made to carry out b, and if A decides that c is what is intended there will be deadlock.

In many cases, of course, such considerations will not apply: it will often be enough to know the set of maximal traces of the use case, without specifying the branching structure. This makes it understandable that the UML standard is written in terms of sets of sequences of actions. However, when describing a use case in text, one has to go out of one's way to avoid specifying where branch points are; it is standard practice that use case descriptions do include this information, because textual descriptions which do so are more natural. Moreover, a use case described using a state chart necessarily has this information, and discarding it would require deliberate identification of structurally different state charts. Therefore it seems likely that our next proposal for a modification to the UML standard may be acceptable: we suggest that "a use case specifies a *process, which determines* a set of sequences of actions".

The set of sequences of actions so determined is the set of maximal traces of the process. We are using the term "process" independently of any particular process algebra; we could use "labelled transition system", "automaton" (but *not* "finite state automaton") or "(restricted) state chart" instead.

2.3 Formalisation and Discussion

Definition 1. *A use case* $u = (S, L, \rightarrow, S_0)$ *consists of:*

- *a (possibly infinite) set* S *of states*
- *a (possibly infinite) set* L *of labels*
- *a transition relation* $\rightarrow \subset S \times L \times S$*; as usual we write* $s \xrightarrow{l} t$ *rather than* $(s, l, t) \in \rightarrow$
- *a (possibly infinite) set of initial states* $S_0 \subseteq S$

We will use the usual variants on the notation, writing for example $s \rightarrow t$ *when there is some* l *such that* $s \xrightarrow{l} t$*, and* $s \not\rightarrow$ *or equivalently* $s \in \mathit{final}(u)$*, when there are no* l*,* t *such that* $s \xrightarrow{l} t$*. We write* $l_1 \ldots l_n \in \mathit{sequences}(u)$ *iff there exist* $s_0, \ldots s_n$ *such that* $s_0 \in S_0$ *and* $s_0 \xrightarrow{l_1} s_1 \ldots \xrightarrow{l_n} s_n \not\rightarrow$

For technical reasons we will insist that $S_0 \cap \mathit{final}(u) = \emptyset$*, that is, that every start state has some transition.*

Specifying a use case. We deliberately do not concern ourselves with how the components of a use case are given, because there are many choices that will be appropriate in different circumstances. The choice, more than our underlying LTS structure, will determine what computation it is possible for a tool to carry out concerning a use case and with what complexity. For example, if it is possible to abstract away from data to the extent that the sets S and L are finite, the whole range of model-checking techniques becomes available. More usually, these sets will be infinite (or extremely large, depending for example on whether one models with real integers or machine integers) but structured according to the structural and behavioural features of the use case, and ultimately of the entity and the actors.

Nevertheless, any way of giving the components should be expected to satisfy a sanity condition that the state of the use case "contains no more information than" the states of the participants in the use case. Precisely:

Definition 2. *Let $u = (S, L, \rightarrow, S_0)$ be a use case for an entity with statespace E in which the entity communicates with actors with statespaces $A_1 \ldots A_n$.*[2] *A* quotient map *of u is a surjective function*

$$h : E \times A_1 \times \ldots A_n \rightarrow S.$$

A specification formalism should include such a quotient map for each use case. Most simply, if the statespace of the use case is constrained to be written in terms of a subset of the attributes of the entity and actors involved, the obvious induced quotient map might be assumed.

Why maximal traces? We have chosen to restrict attention to finite sequences of actions, where no further action is possible, because there is an assumption running through the UML standard that the sequences of actions that comprise a use case are finite. This does not imply that the process needs to be a tree, or even acyclic. We could restrict attention to acyclic processes, which would obviously facilitate reasoning about total correctness. However to impose such a restriction in general would seem unreasonable: there is no obvious objection to reaching the same state by two different routes, or even several times. Considering only maximal traces, in this way, does limit the usefulness of use cases for describing continuing behaviour; but this is already a recognised weakness of use cases. This formalisation might provide a basis for remedying the weakness, but this is beyond our scope here.

Specifying branching structure. As we have argued, branching is a normal feature of use case descriptions. It is normal, however, for the conditions under which the branches are taken to be underspecified at this stage; the precise conditions may not be known, or may depend on data of the system and/or the actors which the specifier does not choose to model. There is a style of specification (and Catalysis' recommendations, for example, are close to it) in which the specifier would add "conceptual" attributes to system and actors and write conditions in terms of these; but it is far from universal. For simplicity, therefore, we will model choices as simple non-determinism in an LTS.

2.4 Use Cases as (Pre- and) Post-Conditions

Pre- and post-conditions (hereinafter PPCs) are suggested in the UML standard as one way of describing a use case, and in the Catalysis method they are the main such method. Evidently they are not sufficient for recording intended contracts between the system and its actors which are more complex than a simple

[2] Notice the use of our restriction that there is at most one instance of a given actor communicating with a given use case: this enables us to confuse actor instances with actors with impunity, and we shall continue to do so

relation between the states before and after an instance of the use case occurs – they provide no way to talk about the nature of the interaction between the system and its actors. Thus two implementations of a use case could in general satisfy the same PPC, but not be substitutable for one another from the point of view of an actor, if, for example, one implementation expected two inputs from the actor where the other expected three. (In Catalysis use cases are identified with (joint and/or abstract) actions; use cases are decomposed into sequences (sic) of smaller use cases/actions. This is an interesting approach; however, in UML use cases and actions are quite distinct concepts, so we will not consider it further here.)

PPCs are, however, a very convenient way to specify use cases which are simple, and/or which should be specified at a very high level. How does this view interact with the others?

Following Catalysis, we consider PPCs written in terms of conceptual attributes of the system *and the actors*: recall that this is the same information which determines our proposed states of use cases via quotient maps.

Definition 3. *Let $u = (S, L, \to, S_0)$ be a use case for an entity E with actors $A_1 \ldots A_n$, and let $h : E \times A_1 \times \ldots A_n \to S$ be its quotient map.*[3]

Let a PPC be given as a relation $PP \subseteq (E \times A_1 \times \ldots A_n) \times (E \times A_1 \times \ldots A_n)$, where if p fails the pre-condition $(p, q) \not\in PP$ for any q; we write p fails pre(PP). Then

1. *PP and u are* compatible *if PP respects h, in the sense that whenever $h(p) = h(p')$ and $h(q) = h(q')$ we have $(p, q) \in PP \Leftrightarrow (p', q') \in PP$.*
2. *u* satisfies *PP iff both*
   a) *PP and u are compatible, and*
   b) *whenever $s_0 \in S_0$ and $s_0 \xrightarrow{l_1} s_1 \ldots \xrightarrow{l_n} s_n \not\to$ and $h(p) = s_0$ and $h(q) = s_n$, we have either $(p, q) \in PP$ or p fails pre(PP).*

Intuitively, to say that a use case and a PPC are compatible is to say that the PPC uses no more state than is recorded in the use case. If this is not the case, then the question of whether the use case satisfies the PPC is nonsensical.

Notice that in the simplest case, where the state of a use case is determined by giving values for a subset of the attributes of the entity and its actors, a PPC written in terms of that same subset of attributes will automatically be compatible with the use case, as expected.

We shall later need the following:

Definition 4. *Let $u = (S, L, \to, S_0)$ be a use case for an entity E with actors $A_1 \ldots A_n$, and let $h : E \times A_1 \times \ldots A_n \to S$ be its quotient map. The* induced pre- and post-condition of *u, written $PP(u)$, is*

$$\{(p, q) \in (E \times A_1 \times \ldots A_n) \times (E \times A_1 \times \ldots A_n) : h(p) = s_0 \in S_0 \text{ and } \exists s_0 \to s_1 \ldots \to s_n \not\to \text{ such that } h(q) = s_n\}$$

Lemma 1. *Any use case satisfies its induced pre- and post-condition.* □

[3] by a slight abuse of notation we use E both for the entity and its state space, etc.

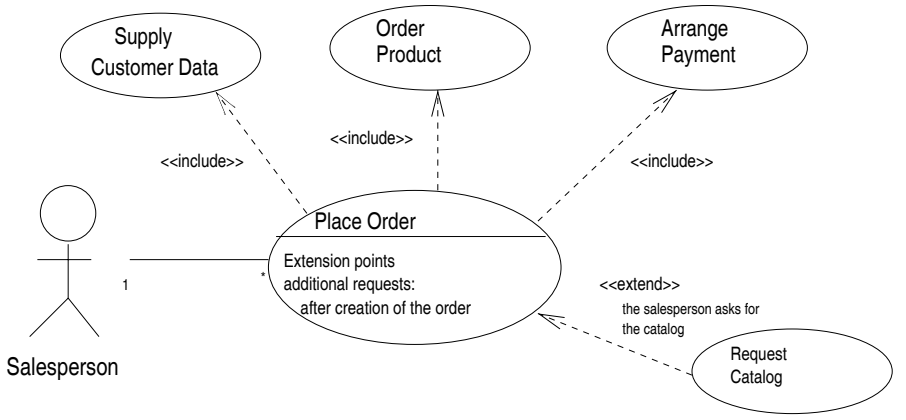


Fig. 3. ≪include≫ and ≪extend≫ dependencies between use cases: copied from [7] (3-93)

3 ≪Include≫ Dependency between Use Cases

The intention of the ≪include≫ relationship is to show subtasks as separate use cases; for example, in order to show that certain subtasks are shared by ≪include≫ing them in several use cases, or to demonstrate the use of a pre-existing component to carry out a subtask. The most relevant passage is:

> An include relationship between two use cases means that the behavior defined in the target use case is included at one location in the sequence of behavior performed by an instance of the base use case. When a use-case instance reaches the location where the behavior of an another use case is to be included, it performs all the behavior described by the included use case and then continues according to its original use case. This means that although there may be several paths through the included use case due to e.g. conditional statements, all of them must end in such a way that the use-case instance can continue according to the original use case. One use case may be included in several other use cases and one use case may include several other use cases. The included use case may not be dependent on the base use case. In that sense the included use case represents encapsulated behavior which may easily be reused in several use cases. Moreover, the base use case may only be dependent on the results of performing the included behavior and not on structure, like Attributes and Associations, of the included use case.

([7] 2-126: a subset of this information also occurs on 2-120)

We must take care over the interpretation of "the included use case may not be dependent on the base use case". In [10] it is stated that the included use case has access to the attributes of its base, which apparently contradicts the

lack of dependence; but indeed, it is not clear in what sense an included use case can *have* a result if it is not allowed to affect the state of its parent! We interpret the lack of dependence as meaning simply that the included use case makes sense, that is, is a valid use case, in its own right. Now as an included use case can define its own attributes as well as having access to those of its parent, we conclude that its quotient map should induce a possibly finer equivalence on the state of the entity and actors than its parent did.

Definition 5. *A use case* $v = (S', L', \to', S'_0)$ *with quotient map* h_v *is* suitable for inclusion in $u = (S, L, \to, S_0)$ *with quotient map* h_u *if*

1. h_v *is finer than* h_u*; that is,* $h_v(p) = h_v(q) \Rightarrow h_u(p) = h_u(q)$*;*
2. $|h_v(h_u^{-1}(s)) \cap S'_0| \leq 1$ *for all* $s \in S$*.*

We write h_{uv} *for the unique surjective map* $S' \to S$ *such that* $h_{uv} \circ h_v = h_u$

In terms of attributes, condition 2 amounts to saying that if the included use case defines a new attribute, then given values for the attributes already present in the base use case there must be a single ("default") value for the new attribute. This avoids the need to invent data for the extra attribute(s). If there are no new attributes, that is, if $h_u = h_v$, the condition is vacuous.

We interpret the instruction that the base use case must depend *only* on the result of the included use case as meaning that the base use case must be describable in terms which allow the substitution of any use case with the same induced PPC as the intended one. We restrict the circumstances under which inclusion is allowed, for convenience and to reflect the intention that ≪include≫ (unlike ≪extend≫) is used for unconditional, once-only inclusion of behaviour. Thus we define:

Definition 6. *A use case* $u = (S, L, \to, S_0)$ *with quotient map* h_u *includes use case* $v = (S', L', \to', S'_0)$ *with quotient map* h_v *via Source* $\xrightarrow{I}$ *Target if*

1. v *is suitable for inclusion in* u*;*
2. $I \in L \setminus L'$*;*
3. *Source, Target,* S_0 *are pairwise disjoint subsets of* S*;*
4. $h_u(p) \xrightarrow{I} h_u(q)$ *is a transition in* u *exactly when all of:* $(p, q) \in PP(v)$*,* $h_u(p) \in$ *Source,* $h_u(q) \in$ *Target hold;*
5. *any element of sequences*(u) *includes at most one occurrence of* I*;*
6. *the only transitions out of Source or into Target are those labelled* I*.*

Such a use case depends only on the result of a suitable v, not on its details. To get a fully described use case including v itself we replace the source-target gap by the actual behaviour of v, identifying source states with the appropriate start states of v and target states with the appropriate final states; precisely:

Definition 7. *If u includes v via Source $\xrightarrow{I}$ Target then the composite use case $include(u, v, I) = (S'', L'', \to'', S_0'')$ where*

1. $S'' = S \setminus Source \setminus Target \dot\cup S'$
2. $L'' = L \setminus \{I\} \dot\cup L'$
3. *For s and t in S'', $s \to'' t$ iff one of the following holds:*
   - $s \to t$ *in* u *(note that by insisting s, t are in S'' we automatically discard transitions to Source or from Target)*
   - $s \to s' \in Source$ *in* u *and* $t \in h_v(h_u^{-1}(s')) \cap S_0'$ *(that is, t is a start state of v corresponding to a source in u)*
   - $t' \in Target$ *and* $t' \to t$ *in* u *and* $t' = h_{uv}(s)$ *(that is, s is a final state of v corresponding to a target in u)*
   - $s \to' t$ *in* v
4. $S_0'' = S_0$

The easy result that relates this process view to the trace inclusion view found in the standard is:

Lemma 2. *If $l_1 \dots l_n \in sequences(include(u, v, I))$, then either*

- $l_1 \dots l_n \in sequences(u)$*; or*
- *for some $1 < m \le p \le n$ we have $l_1 \dots l_m I l_p \dots l_n \in sequences(u)$ and $l_{m+1} \dots l_{p-1} \in sequences(v)$.*

□

We make no formal restriction on when the first case can occur, but to stay within the spirit of the ≪include≫ relation this should indicate that something exceptional happened "before" the "inclusion point" of v. Because use cases are allowed to include exceptional or erroneous cases, in which the use case instance might be "aborted" before v becomes relevant, it does not seem sensible to take literally the view that every trace of the composite use case should include a trace of v. One could make this artificially the case by writing v to include a trivial "not applicable" trace to be included at the end of any trace of u on which v is never truly reached, but this seems pointless.

The ≪extend≫ relationship between use cases is similar; it is more complex but raises no substantially new issues. For reasons of space we omit a detailed treatment.

4 Generalisation of Use Cases

This is the most controversial of UML's relationships between use cases. (Cockburn writes, for example "In general, the problem with the *generalizes* relation is that the professional community has not yet reached an understanding of what it means to subtype and specialize behavior..." [4] p241.) The Notation Guide contains no example of generalisation of use cases, and says only the "A generalization from use case A to use case B indicates that A is a specialization of B" (3-92). The example pictured above is taken from [10] and is typical of the way the relationship is used: the idea is that the child use case should be a more fully described version of its parent. The Semantics Guide appears more helpful:

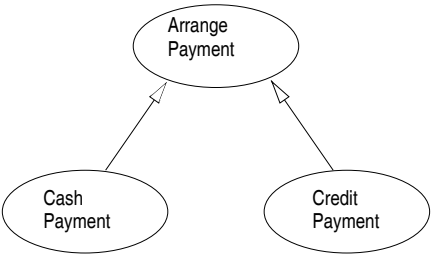


Fig. 4. Generalisation of use cases, from [10]

> Generalization between UseCases means that the child is a more specific form of the parent. The child inherits all Features and Associations of the parent, and may add new Features and Associations.

(2-120)

> A generalization relationship between use cases implies that the child use case contains all the attributes, sequences of behavior, and extension points defined in the parent use case, and participate [sic] in all relationships of the parent use case. The child use case may also define new behavior sequences, as well as add additional behavior into and specialize existing behavior of the inherited ones. One use case may have several parent use cases and one use case may be a parent to several other use cases.

(2-126)

Trying to combine this with the property which any UML Generalization is supposed to have, namely, that the child GeneralizableElement may be safely substituted for the parent, demonstrates that this description is at least incomplete. To see this, suppose A is an external system, behaving as an actor expecting to interact with an entity E according to a use case u. Presumably A includes code to react correctly to all actions which it may receive from E. Scenarios in the use case u contain a mixture of actions sent from A to E, those sent from E to A, and internal actions which may for example change the state of E. If we replace u by a specialising use case v which is allowed to contain arbitrary new scenarios, there is no guarantee that A will be able to react correctly, or at all, to actions which it may receive as part of the new scenario. That is, v will not automatically be substitutable for u. Let us examine what it should mean for v to be a specialised version of u.

Essentially, the wording of the current standard does not distinguish between two fundamentally different possible relations between u and v, each of which arguably reflects the view that more design decisions have been reflected in the description of v than of u:

1. v is more deterministic than u in what it says about the entity's actions
2. v replaces descriptions of high-level actions by lower-level more detailed descriptions.

Each of these points requires further analysis. (2) is essentially what we did with ≪include≫, though a more flexible treatment would be useful. The main thing to notice about (1) is that it only makes sense if we know which actions are chosen by the entity and which by an actor. So far, we have not needed to make such a distinction, and indeed we pointed out that it is sometimes desirable to describe use cases before such decisions have been taken. From the point of view of an actor for whom v should be substitutable for u, it would be quite reasonable for v to include fewer transitions chosen by the entity, or indeed by other actors, but it would be unreasonable to include fewer transitions chosen by A. Again, A may have been coded to assume that it is permitted to take a certain transition, and might break if v forbade this. From a theoretical point of view the most obviously appealing formalism to use to reflect this question of where choices are made would be the alternating transition systems of [2]; but as remarked before this conflicts with our desire to use minimal machinery – especially as we would need to do at least a little adaptation to deal with the presence of more than the two choice-makers considered there. Perhaps the reactive module work of [1], might help. For now we adapt an approach due to Jackson in [6], and say that *from the point of view of a given actor A*, the actions that make up a use case may be classified into inputs (actions initiated by A affecting E), outputs (actions initiated by E affecting A) and all others (including internal actions by E and communications between E and actors other than A: recall that UML does not permit direct communication between actors, so this list is exhaustive). To what extent does an actor care about more than its own input/output? It presumably does care about the PPC of the use case; after all, this is what specifies what the use case actually achieves. For simplicity in these early explorations, let us assume that u and v have identical label-sets and a bijection f from the start-states of v to the start-states of u.

From the point of view of an actor A who sees the actions of a use case u as FromMe $\dot\cup$ ToMe $\dot\cup$ Other, write $\stackrel{l}{\Rightarrow}$ for $\stackrel{l}{\rightarrow}$ preceded or followed by arbitrarily many Other transitions. Then a candidate definition is:

Definition 8. *$v = (S', L, \rightarrow', S'_0)$ specialises $u = (S, L, \rightarrow, S_0)$, iff the following all hold:*

- *v satisfies $PP(u)$*
- *there is some relation $\preceq \subseteq S' \times S$ such that $s_v \preceq s_u$ implies*
  - *whenever $i \in$ FromMe and $s_u \stackrel{i}{\Rightarrow} s'_u$ is a transition in u, then there is a transition $s_v \stackrel{i}{\Rightarrow}' s'_v$ in v such that $s'_v \preceq s'_u$;*
  - *whenever $o \in$ ToMe and $s_v \stackrel{o}{\Rightarrow}' s'_v$ is a transition in v, then there is a transition $s_u \stackrel{o}{\Rightarrow} s'_u$ in u such that $s'_v \preceq s'_u$;*
  - *$s'_0 \preceq f(s'_0)$ for all $s'_0 \in S'_0$.*

A point that follows on from this is that if u involves several actors, which use cases v are specialisations of u depends on from which actor's point of view we see the relationship. What is the relation that results from insisting that v be a specialisation from the point of view of all (beneficiary) actors? What difference

does it make if actors are assumed capable of cooperation? Can these relations be put in a testing framework? Given appropriate high-level syntax, can they be computed? We have abandoned the total correctness aspects of [6] (because the assumptions seem too strong for our context); can we recover something useful when we want it? For these or some similar definitions, do ≪include≫ and generalisation interact cleanly, for example, is there some sense in which you can be permitted to include a more specialised version of what you expected, and is the resulting composite use case a more specialised version of what you would have had with the more general included use case? This may be too much to hope.

5 Conclusions and Further Work

It is possible, and often pragmatically advisable, to use UML use cases in a completely informal way, to describe the tasks to be carried out with the help of the system. Sometimes, however, we would like to have the option of relating use cases to the design of the system that implements them – or to that of another system that uses them – in a soundly-based, tool-supportable way. Moreover, regardless of whether users of UML choose to describe their use cases in full detail, it seems reasonably to expect that the UML standard's informal explanations of what would be the case if one did, would be formalisable. In this paper we have begun an attempt to provide, using the minimum of formal machinery, such a formalisation. It is surprising (at least to the author), given the modest scope of the present work, how much already has to be done. Directions of future work might include, besides the answering of questions left open here:

High level formalisms for describing use cases. We have deliberately avoided choosing a high-level formalism for defining use cases, preferring to work at the basic level of labelled transition systems or processes. These can for example be defined using any process algebra or any kind of automata that has a labelled transition system semantics; alternatively, they could be defined using (a suitable formalisation of) statecharts. We suggest that it is probably impractical to standardise on one such language within UML, at least at this point; were a formalisation of statecharts to be agreed in UML2.0, this would be an obvious candidate, but others might be more appropriate in different circumstances. It seems more appropriate to use the minimal notion of labelled transition system in the UML standard.

Interference. We should consider interactions and interference between use case instances, both of the same use case and of different use cases. Pace [7]'s statement "This implies that after its performance the entity will in general be in a state in which the sequence can be initiated again.", ordering dependencies between use cases often exist and are sometimes modelled with activity diagrams; how can this be formalised? What is the relationship between the set of all use cases and the entity itself? [7] (2-124) says "The complete set of use cases specifies all different ways to use the entity, i.e. all behaviour of the entity is expressed by its use cases."; in what formal sense can this be true?

Acknowledgements. I am grateful to the British Engineering and Physical Sciences Research Council for funding in the form of an Advanced Research Fellowship, and also to the referees for constructive comments.

References

[1] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.

[2] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, , and Moshe Y. Vardi. Alternating refinement relations. In *Proceedings of the Ninth International Conference on Concurrency Theory (CONCUR 1998)*, number 1466 in LNCS, pages 163–178. Springer-Verlag, 1998.

[3] C. Bolton and J. Davies. Activity graphs and processes. In *Proceedings of IFM 2000*, 2000.

[4] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2001.

[5] Desmond D'Souza and Alan Cameron Wills. *Catalysis: Objects, Frameworks and Components in UML*. Addison-Wesley, 1998.

[6] Paul B. Jackson. Total correctness refinement for sequential reactive systems. In *proceedings of TPHOLs 2000. (13th International Conference on Theorem Proving in Higher Order Logics)*, number 1869 in LNCS, pages 320–337. Springer-Verlag, August 2000.

[7] OMG. *Unified Modeling Language Specification version 1.3*, June 1999. OMG document 99-06-08 available from www.omg.org.

[8] G. Övergaard and K. Palmkvist. A Formal Approach to Use cases and their Relationships. In P.-A. Muller and J. Bézivin, editors, *Proceedings of ≪UML≫'98: Beyond the Notation*, pages 309–317. Ecole Supérieure des Sciences Appliquées pour l'Ingénieur – Mulhouse, Université de Haut-Alsace, France, June 1998.

[9] Gunnar Övergaard. *Formal Specification of Object-Oriented Modelling Concepts*. PhD thesis, Department of Teleinformatics, Royal Institute of Technology, Stockholm, Sweden, 2000.

[10] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1998.

On the Importance of Inter-scenario Relationships in Hierarchical State Machine Design

Francis Bordeleau and Jean-Pierre Corriveau

School of Computer Science, Carleton University, Ottawa, Canada
{francis, jeanpier}@scs.carleton.ca

Abstract. One of the most crucial and complicated phases of system development lies in the transition from system behavior (generally specified using scenario models) to the detailed behavior of the interacting components (typically captured by means of communicating hierarchical finite state machines). This transition constitutes the focus of this paper. Our standpoint is that in order to succeed with this transition, it is not sufficient to grasp the details of individual scenarios, but also to understand the relationships that exist between the scenarios. Our claim is that different types of scenario relationships result in different hierarchical state machine structures. We identify and describe four different types of scenario relationships: containment dependency, alternative dependency, temporal dependency, and functional dependency. We also illustrate how such scenario relationships can be expressed in UML using stereotypes and how they guide the design of the corresponding hierarchical state machines.

1 Introduction

In current object-oriented modeling methodologies [1,2], systems are generally modeled using three apparently orthogonal 'views' namely: structure, behavior, and interactions. Semantically, however, these three views overlap considerably [3,4]. That is, they present much of the same conceptual information from different perspectives and with varying levels of detail. In particular, the behavioral view expresses the complete behavior of each component (or class) of a system in terms of states and transitions, typically organized into hierarchical state machines [5]. Conversely, the interactions view captures how system requirements map onto sequences of responsibilities to be executed by components [3,4,6]. Such sequences are generally referred to as *scenarios*. In UML, such scenarios can be captured at different levels of abstraction using use cases, activity diagrams, sequence diagrams, and collaboration diagrams [7]. In our terminology, we use the term *scenario* to refer to one of the different paths of execution captured within a single *use case*. That is, each scenario forms a specific sequence of responsibilities. We argue in section 2 for the importance of a scenario-driven, as opposed to a use case-driven approach to the design of component behavior.

In a scenario-driven approach to object-oriented modeling [e.g., 8], the scenarios of a system constitute the foundation for the design of the complete behavior of the individual components of this system. That is, whereas each scenario illustrates the behavior of (one or more instances of) a component (viz. class) in a specific context, the state machine associated with this component captures the

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 156-170, 2001.

behavior of that component across all the contexts to which (instances of) it participates. We consider the transition needed to go from scenario models to communicating hierarchical state machine models to be one of the most crucial and complex steps of object-oriented system design. Factors that contribute to the complexity of this transition include:

- **Large number of scenarios.** Typical object-oriented systems are composed of very large sets of scenarios, and each component is usually involved in the execution of many different scenarios. A component's behavior needs to handle each of the scenarios in which this component is involved.
- **Concurrency and interactions between scenarios.** This point follows from the previous one. Scenarios can be concurrent and may interact with each other (e.g., one aborts another or is an alternative to it). This is an aspect of object-oriented systems that makes them particularly complex and difficult to design. More specifically, it is not sufficient to address only the temporal ordering of scenarios. In fact, it is their functional (or equivalently, logical) ordering, and in particular their causal ordering, that must be taken into account in the specification of component behaviors.
- **Scenarios of different types.** Object-oriented systems generally implement scenarios of different types. For example, one may categorize scenarios into those that tackle normal executions ('primary' scenarios), and those that capture alternatives ('secondary' scenarios). Another categorization scheme may focus on the functional aspects of scenarios: control, configuration, service interruption, failure, error recovery, maintenance, etc. Identifying such scenario types may impact on the structuring of component behavior.
- **Maintainability and extensibility of components.** Since most industrial systems have a long lifecycle, it is very important to build system components so that they can be easily maintained and extended. Thus, it is not sufficient to define component behaviors that satisfy the current requirements. It is also desirable to define them so that they facilitate future modifications. Indeed, we contend that the structuring of the behavior of a component is one of the most important factors contributing to component maintainability and extensibility. This constitutes an important nonfunctional requirement that must be considered by designers.

The transition between scenario models and hierarchical state machines constitutes the focus of our work [4,9]. Our standpoint is that in order to successfully proceed from scenarios to state machines, it is not sufficient to grasp the details of individual scenarios, but also to understand the relationships that exist *between* these scenarios. We believe that understanding the relationships that exist between a set of scenarios that are to be implemented in a communicating hierarchical state machine model is one of the fundamentals steps in the design of complex object-oriented systems.

Our thesis is that different types of scenario relationships result in different hierarchical state machine structures [*Ibid.*]. More precisely, for a component *x*, the semantic relationship between two scenarios S1 and S2 to which *x* participates should guide the designer in integrating the behavior associated with *x* in S1 with the behavior of *x* in S2. Indeed, we have proposed elsewhere [*Ibid.*] a hierarchy of *behavior integration patterns* used to provide a systematic approach to the transition from scenarios to hierarchical state machines.

In this paper, we sketch out the scenario relationships at the basis of these patterns. We first summarize, in section 2, our pattern-based approach to the transition

from scenarios to state machines. We then introduce, in section 3, a simple example used to illustrate the scenario relationships from which this transition proceeds. Each of these relationships is defined and overviewed in section 4. We review other related approaches in section 5 and discuss the issue of automation. We conclude with a brief discussion of some of the advantages of our approach.

2 Proposed Approach

In order to design the (possibly hierarchical) state machine of a complex component, the modeling strategy we propose draws on both the relationships and details of scenarios. The exact notation used for the specification of scenarios (e.g., message sequence charts (MSCs) [10], sequence diagrams [1,7]) does not play a significant role in the use of our integration patterns. However, we remark that, except for Use Case Maps (UCMs) [4,6], few notations *explicitly* capture *inter-scenario* relationships. And yet we have found that such explicit relationships do simplify the application of the integration patterns we suggest.

Our thesis, we repeat, is that, for some component *x*, the integration of a new scenario S1 (in which *x* participates) into the existing state machine *f* of *x* must depend on the pair-wise relationships existing between S1 and the scenarios already handled by *f*. This aspect of our work will be discussed in section 4. For now, we want to summarize the overall approach we are proposing for going from scenarios to state machines. This approach, summarized in Figure 1, consists of three steps:

First, the designer must capture scenarios and their inter-relationships. Recall that a use case may capture a multitude of different scenarios, that is, a multitude of different paths of execution. Thus, a relationship between two use cases expresses a semantic dependency between sets of scenarios, rather than between specific scenarios. Our thesis is that specific inter-scenario relationships play a major role in the design of the hierarchical state machines of the components involved in these scenarios. The exact nature of these relationships is of little import in most existing UML-based modeling processes. In contrast, in the first step of our approach, we are recommending that a designer start by identifying the pair-wise relationships between scenarios. From our standpoint, the exact classification of these relationships is not essential to the application of our strategy; only that each inter-scenario relationship has a specific impact on the design of the state machines associated with the relevant scenario components. In other words, the set of inter-scenario relationships introduced in section 4 is not to be taken as sufficient or complete, but merely as representative. Indeed, we will briefly demonstrate, in that section, that each of the four inter-scenario relationships we put forth has a direct impact on the organization of the state machines of the relevant components.

In UML, scenarios are first captured by means of use cases. The latter are not meant to exist in complete independence from one another: a use case diagram is to capture their relationships, which take the form of unidirectional associations. User-defined stereotypes can be specified for these associations (as for any associations in UML). UML currently proposes four built-in stereotypes [1,7] for use cases, none for their relationships. Moreover, use case relationships are not our primary concern. Instead, we focus firstly on scenario relationships. But use case and scenario relationships clearly overlap semantically.

Second, each use case is refined into a set of diagrams (e.g., sequence diagrams, MSCs) detailing the interactions between the components involved in the scenarios of that use case. In other words, the individual scenarios are now re-expressed in the form of what we will call generically *interaction diagrams*. (The transition between use cases and interaction diagrams may involve several design decisions not discussed here, as well as the use of other models such as class diagrams.) From each such diagram, the designer extracts a state machine for each component that participates in that scenario. We call such state machines *role state machines* for they describe behavior that must be implemented by a component to play a specific role in a specific scenario. From the viewpoint of a single component, we end up with a set of role state machines to integrate into the single state machine of that component.

The third step of our proposal consists in the integration of the role state machines of a component with the existing state machine *f* of that component. In order to integrate a role state machine of scenario S1 with *f*, the designer must establish the pair-wise relationships that exist between S1 and the scenarios already handled by *f*. Each relationship, we repeat, must suggest a specific behavior integration pattern [9].

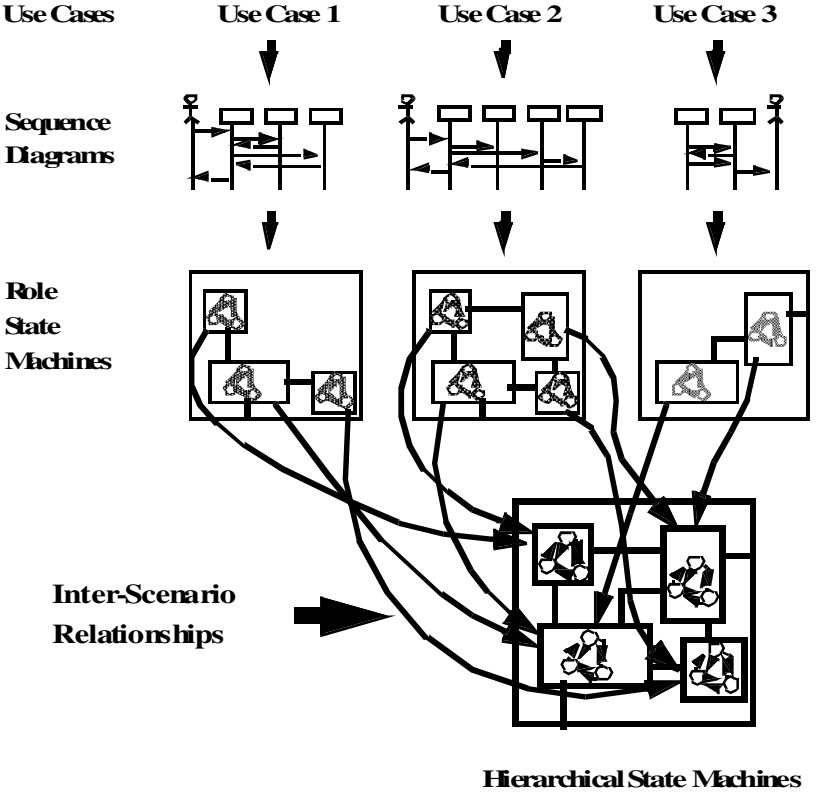


Fig. 1. From Scenarios to Hierarchical State Machines

Fig. 2 gives our current hierarchy of behavior integration patterns (which is discussed at length elsewhere [4]). This hierarchy includes a set of general patterns that can be used as a starting point to define a more complete catalogue of integration patterns in specific development contexts. The proposed 'scenario interaction' design patterns proceed from the temporal and logical relationships that may exist between a role to integrate and the roles already handled by the behavior of a component. For example, with respect to such relationships, two roles of a component (or equivalently their corresponding scenarios) may be mutually exclusive, or one may abort another, or wait for another, or follow another, etc. Conversely, the Mode-Oriented Pattern is one that addresses the structuring of a component's behavior. More specifically, it puts forth an overall hierarchical architecture for the design of a component whose behavior can be thought of as a sequencing of modes.

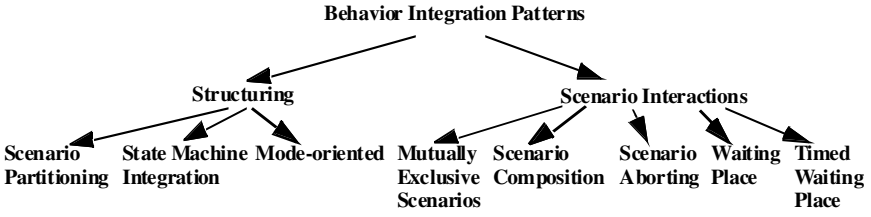


Fig. 2. A Hierarchy of Behavior Integration Patterns

Finally, our catalogue also offers two process patterns: the Scenario Partitioning Pattern and the State Machine Integration Pattern. Such process patterns aim at guiding the designer through the typical steps to be followed when grouping or ungrouping sets of behaviors within a same component. For example, the State Machine Integration Pattern spells out the four steps to use when trying to decide how and where to integrate a new role into an existing hierarchical state machine. (See [4] for more details.)

3 A Simple Example

In this paper, we use a simple Automatic Teller Machine (ATM) system to illustrate the different scenario relationships we introduce in the next section. This ATM system is a conventional one that allows for withdraw, deposit, bill payment, and account update.

The ATM system is composed of a set of geographically distributed ATMs and a Central Bank System (CBS), which is responsible for maintaining client information and accounts; and for authorizing and registering all transactions. Each ATM is composed of an ATM controller, a card reader, a user interface (composed of a display window and a keypad), a cash dispenser, an envelope input slot (used for deposit and bill payments), and a receipt printer. The ATM controller is responsible for controlling the execution of all ATM scenarios, and for communicating with the CBS. For simplicity, we will only consider here the behavior of the ATM Controller.

In this paper, we consider only the following scenarios:

- A start-up scenario that describes the steps required for bringing the system to its operational state. The start-up scenario requires two input messages from the system administrator: a start-up message that triggers the configuration of the ATM, and a start message that starts the ATM. The start message can only be input after completion of the configuration.
- An initial configuration scenario that describes the sequence of steps that are required to properly configure an ATM. These steps include the configuration of each components of the ATM system, and the establishment of a communication with the CBS.
- A *Transaction* scenario that captures the sequence of responsibilities common to all transactions. It includes reading the card information, verifying the PIN (Personal identification Number), getting the user transaction selection, executing the required transaction, printing a receipt, and returning the card.
- One scenario for each of the different types of transactions offered by the ATM system: withdraw, deposit, bill payment, and account update. Each scenario gives the details of a specific transaction, as well as a set of relevant alternatives.
- A shutdown scenario that describes the steps to be carried out when closing down the ATM. The shutdown steps include turning off the different ATM components, and terminating communication with the CBS.

4 On Scenario Relationships and Integration of Behavior

We are suggesting that the exact inter-scenario relationship between two scenarios determine how these scenarios are to be integrated into a hierarchical state machine [4,9]. Our goal here is not to be exhaustive, but instead to demonstrate the importance of each of the four inter-scenario dependency relationships we introduce namely: containment, alternative, temporal and functional.

4.1 Containment Dependency

A containment dependency exists between a scenario S1 and a scenario S2, if S2 is used in the description of S1. Examples of this type of relationship include stubs in UCMs [4,6], and the "uses" relationship defined by Jacobson [12]. This relationship is essential for the development of complex systems as it allows for the recursive decomposition (or composition) of scenarios into other scenarios.

In the ATM example, a containment dependency exists between the Transaction scenario and each of the scenarios corresponding to the steps of that scenario, as illustrated in Fig. 3.

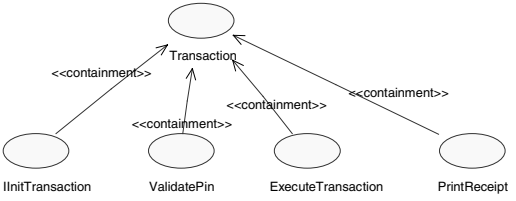


Fig. 3. The Containment Stereotype

At the hierarchical state machine level, such dependencies can be captured explicitly through a simple strategy, namely, defining a state machine for each contained scenario, and placing such state machines as substates of the state corresponding to the container scenario. In this paper, hierarchical state machines are described using the UML notation [1]. In our example, we obtain the following state machine for the **transaction** state (which is to handle a transaction) of the ATM controller (whose top-level state machine will tackle other aspects of the controller, such as starting up and shutting down, as shown in Figure 8 later).

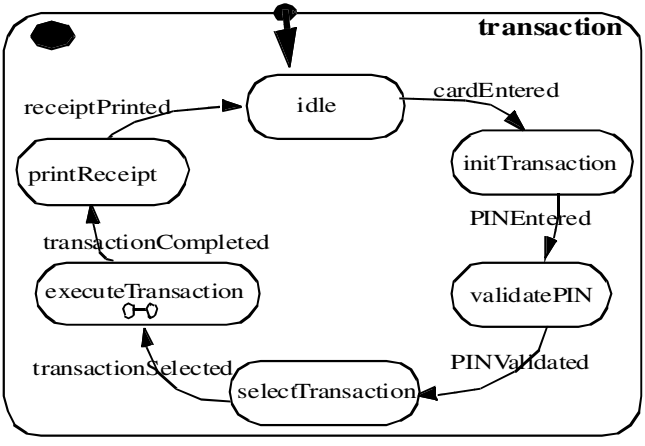


Fig. 4. A Hierarchical State Machine for the Steps of Using an ATM

The containment dependency is a simple inter-scenario relationship that appears to be semantically sufficient for the hierarchical organization of the state machine of a component. In particular, we believe it is unnecessary to introduce a more complex inter-scenario relationship that would somehow involve the concept of inheritance (as Jacobson's *extends* relationship [12], now called *refines* in UML [7]). The semantics of such an "extends" is somewhat problematic (see [11]) and, more importantly from our standpoint, would not directly bear on the design of a state machine.

4.2 Alternative Dependency

A use case outlines "normal mainline of behavior as well as variants such as alternate paths, exception handling [, etc.]" [8]. When a use case is broken down into individual scenarios, it is essential that these alternate paths be semantically tied to the mainline of behavior. For this purpose, we use an inter-scenario relationship we call the *alternative dependency*. In a use case diagram, this relationship is captured using a stereotype of the same name. For example, in our ATM, the *Transaction* scenario must be linked to its alternative, the *Invalid PIN* scenario. To capture this in a use case diagram, it is simplest to have a use case for each scenario and tie these scenarios with the appropriate stereotyped association, as shown in Fig. 5.

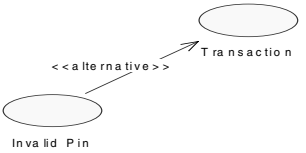


Fig. 5. The Alternative Stereotype

Most importantly, we claim that an alternative dependency suggests a specific strategy to integrate the behavior of a component in the alternate scenario with the behavior of this component for the mainline scenario. The solution typically simply consists in associating transition(s) for the alternate scenario to the state machine handling the mainline behavior. From the existing transaction state machine of the ATM controller (left side of Figure 6), adding the handling of the Invalid Pin scenario simply consists in adding the PINInvalid transitions (as shown in bold on the right side of Figure 6).

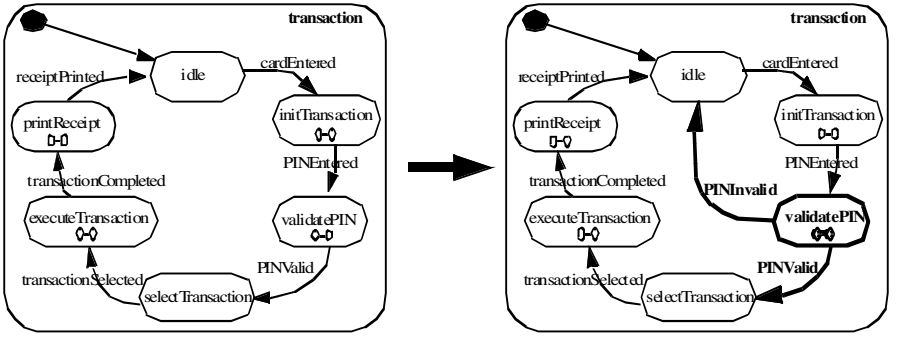


Fig. 6. Integrating the Invalid Pin Scenario into the Transaction Scenario of the State Machine of an ATM

Finally, we remark that, semantically, we found it much easier to deal with alternatives using scenarios, that is, with respect to paths of execution (i.e., sequences of responsibilities), than by referring to use cases. The problem lies in the fact that a use case does not necessarily have a single point of termination, whereas a scenario does. Thus, having a use case as an alternative to another can be quite complicated to understand. (See [11] for further discussion of the semantic difficulties of use-cases.) Conversely, a scenario is an alternative to another if they have a common starting point but distinct end points, much like the different paths of a related path set in Use Case Maps [6].

4.3 Temporal Dependency

This third type of inter-scenario relationship is the strongest we propose from a semantic viewpoint for it allows several specific specializations, which pertain to the temporal ordering of the scenarios (*e.g.*, one scenario *excludes*, *waits for*, *aborts*, *rendezvous* or *joins* another, two scenarios run concurrently, etc.).

As a first example, we want to capture the fact that the shut down scenario must follow the start up scenario (as captured in the partial use case diagram of Fig. 7.).

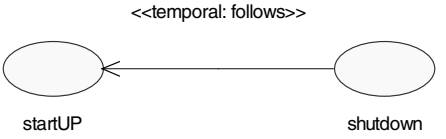


Fig. 7. The Temporal: follows Stereotype

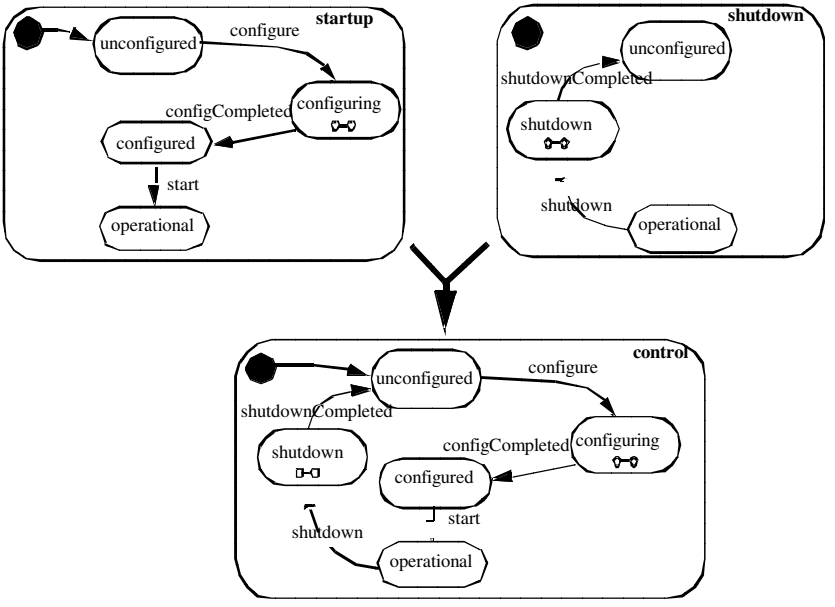


Fig. 8. Control State Machine

The consequences of this relationship on the design of the state machine of the ATM controller are quite direct: each scenario corresponds to a role state machine. These machines must be integrated in such a way that the end of the first one *enables* (via a state, or a precondition, etc.) the start of the second. Here, for example, the end state of *startUp* is the starting state of *shutDown*. The result of integrating the *startUp* and *shutDown* scenarios into a *control* state machine is illustrated in Fig. 8.

The pair-wise specification of such temporal relationships may be cumbersome. For example, we would like to model the fact that the *Withdraw*, *Deposit*, and *Pay Bill* scenarios contained in the *Transaction* scenario are to be mutually exclusive. Though this could be achieved through a multitude of associations, we instead suggest an alternative strategy that draws on another key UML concept, that of a package [see [1, 7]). Here, the “mutual exclusion” temporal

relationship may be captured by simply putting the different relevant scenarios in a package, which is itself assigned the stereotype *MutualExclusion*.

Again, the key point to understand is that this temporal *MutualExclusion* inter-scenario relationship has direct impact on the design of the corresponding hierarchical state machine. In our example, this relationship suggests the use of the *Mutually Exclusive Scenario* behavior integration pattern of Figure 2. This pattern organizes the different mutually exclusive scenarios as states (here, of the ATM controller) that are accessed through a choice point (see [7]), as illustrated in Fig. 9.

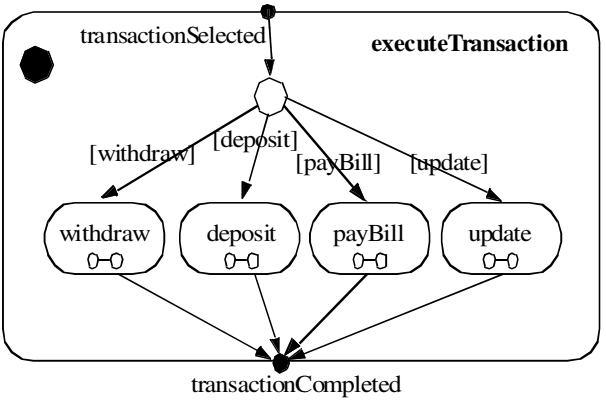


Fig. 9. Mutual Exclusion Example

4.4 Functional Dependency

This last relationship we propose here is used to capture the coexistence of two or more scenarios inside a same conceptual (or logical) regrouping called a *cluster*. This cluster corresponds to a specific 'aspect' of the system. Our design experience in real-time event-driven systems suggests regroupings such as control, configuration, communication, error recovery, normal operation, etc. For example, the *start-up* and *shutdown* scenarios are both part of the *control* cluster of the ATM system, and the deposit and withdraw scenarios are both part of the *operational* cluster. Such clusters can be explicitly captured in a use case diagram through the use of packages. More specifically, a set of scenarios (expressed as individual use cases) can be regrouped into a package. In this case, as with mutual exclusion, the use of a package saves us the trouble of defining a multitude of associations between the scenarios.

The functional dependency is the weakest form of dependency between scenarios as it mostly rests on the viewpoint of the designer. But, from a top-down perspective, establishing early functional dependency between scenarios corresponds to the well-accepted design technique called separation of concerns, which is often required for scalability reasons: each aspect (i.e., group/package of scenarios) may be further decomposed into smaller sets of related scenarios. For example, a subset of the control scenarios may be related to system restart (with different types of restart), while another subset may be relate to error handling (including both error detection and error recovery) and yet another may be related to preparing the system for

reconfiguration. Furthermore, the set of configuration scenarios may be composed of several distinct subsets of scenarios, each related to a different type of system configuration. And, in our ATM example, even the set of 'normal' operation scenarios (i.e., withdraw, deposit, etc.) may be composed of different types of system functionality. Consequently, even apparently simple component behavior can become rather complex to design when having to address issues such as robustness, reliability and extensibility. And thus, the importance of capturing functional dependency.

Furthermore, as usual, our claim is that such an inter-scenario relationship has direct impact on the design of the relevant state machine(s). More specifically, in order to integrate scenarios contained in different clusters, we use the *State Machine Integration* pattern of 0. This pattern (see [9] and [4] for details) suggests that each aspect in which a component participates be given its own hierarchical state in the behavior of that component. For example, after analyzing the overall set of ATM scenarios, scenarios have been partitioned into four subsets: normal operation, control, maintenance, configuration, and secure communication. The result of the partitioning is given in Fig. 10. (For readability purposes, only some of the scenarios are shown in this figure. See [4] for more details.)

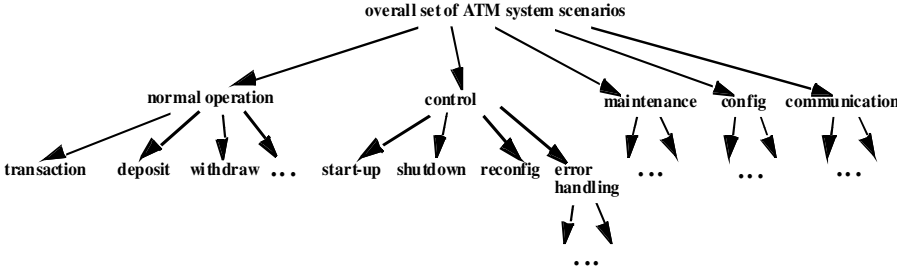


Fig. 10. Scenario clusters for the ATM

Then, state machines are defined on a per cluster basis. For example, in the current case, we separately defined a state machine for the control (Fig. 8) and transaction (Fig. 4) cluster. The final step of the *State Machine Integration* pattern consists in integrating the state machines related to the different clusters in a single hierarchical state machine. This last step is illustrated in Fig. 11 where the control state machine and the transaction state machine (top part of Fig. 11) are integrated into a single hierarchical state machine given in the bottom part of Fig. 11: the transaction state machine becomes the state machine of the operational state of the control state machine. For this purpose, the operational state of the control state machine is modified to become a composite state.

The actual integration is more complex than what is suggested by this figure. When integrating the state machines corresponding to different scenarios, it is essential to verify (e.g., through pre- and post-conditions) that the inner workings of each scenario are preserved.

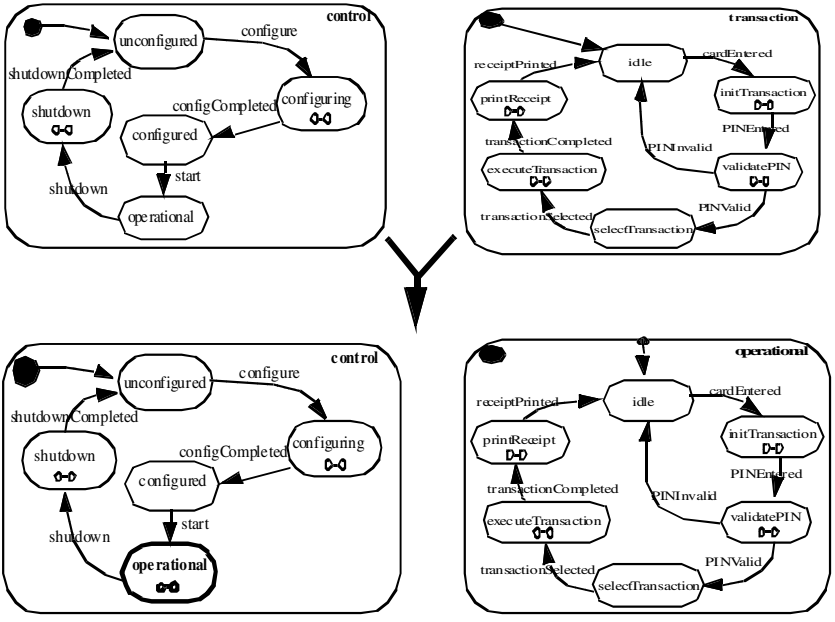


Fig. 11. Scenario Integration Pattern

5 Comparison with Other Approaches

In the current literature, several papers address the problem of defining state machines from different types of interaction diagrams (i.e., scenario models). But most, if not al, ignore scenario inter-relationships. In [13], Haugen discusses a general approach for making the transition between an MSC model [14] and an SDL model [15]. Other papers ([16, 17, 20]) propose an automatic synthesis approach. That is, they aim at explaining how to automatically generate state machines from different types of interaction diagrams. For example, the work of Koskimies and Makinen [16] focuses on the definition of a transition between interaction diagrams and flat (i.e., non-hierarchical) state machines. It defines a synthesis algorithm for the automatic generation of such state machines from a set of sequence diagrams. Leue and Mehrmann [17] offer a similar approach between MSC'96 diagrams [18] and ROOM [19] hierarchical state machines. Also, Harel and Kugler [20] present an automatic synthesis approach centered on the use of live sequence charts (LSCs), a specification technique they define as an extension of interaction diagrams. LCSs are to be used to specify both mandatory and forbidden scenarios.

In contrast, our work centers on the design of hierarchical finite state machines from scenarios *and their relationships*. We believe that automation is ill-suited for such a task. Let us elaborate. Contra synthesis approaches, we recall that, generally, scenario models are incomplete. Thus, the corresponding state machines may typically have to be augmented with states and/or transitions in order to account for behavior not explicitly modeled in the scenarios. In other words, most often, state machines are

not semantically equivalent to the scenarios from which they proceed, but in fact constitute 'semantic extensions' to these scenarios. But the very nature of state machines makes them difficult to modify: for example, one new transition may generate indeterminism. Thus, we repeat, it is essential to immediately structure component behavior for extensibility. And, in our opinion, this is not achieved in synthesis approaches. That is, we believe that an automatically generated state machine is seldom 'semantically customizable'. This is especially true for flat state machines.

The generation of a flat state machine from some algorithm is undoubtedly useful for activities such as validation and verification. However, we find it absolutely crucial to use hierarchical state machines for designing components: hierarchical state machines not only enable iterative development of the behavior of a component, but also promote its understandability, testability and scalability. Whereas a flat state machine, regardless of its number of states, is suitable for exhaustive path analysis, hierarchical state machines allow for the clustering of different behavioral facets of a component (often referred to as roles) into distinct hierarchical states of this component. Such a separation of concern is required for the understanding of complex behavior, and for its evolution, two essential characteristics of a good design. Put another way, we need hierarchical states to decouple clusters of states from each other in order to i) understand their corresponding roles and ii) ease the evolution of each such role (in particular with respect to behavior not explicitly captured in scenario models).

Subsuming this argumentation, we find our conviction that component design is an act of creation guided by a multitude of decisions typically lost in the automatic generation of a (typically flat) state machine from some other (often incomplete) model or specification.

The idea of patterns ([22, 23, 24]) has been put forth in several subfields of software engineering (e.g., analysis, architecture, design, testing) as a means to document typical sets of decisions. Work on behavioral patterns (e.g., Gamma et al. [23]) has mostly focus on inter-component behavior. Indeed, few researchers have focused on patterns for the design of the behavior of a component. Douglass's work on dynamic modeling (chapter 12 of [25]) is a noticeable exception: he is concerned with "useful ways to structure states rather than objects" (Ibid., p.588). And he clearly advocates hierarchical state machines to do this: "many of the state patterns use orthogonal components to separate out independent aspects of the state machine." (Ibid., p.589).

The 18 state behavior patterns put forth by Douglass form a catalogue of typical behaviors for a component. Let us consider, for example, the Polling State Pattern. "Many problems require active, periodic data acquisition. The object processing this data often has states associated with the problem semantics that are unrelated to the acquisition of data... The Polling State Pattern solves this problem by organizing the state machine into two distinct orthogonal components. The first acquires the data periodically. The second component manages the object state... This allows data handling to scale up to more complex state behavior." (Ibid., p.593).

Such a description, and in particular the need for hierarchical states and the goal of conventionalizing behavior, emphasizes the closeness of Douglass's work to ours. But there is a significant difference between our approaches: his catalogue of patterns proceeds from his identification of a set of typical behaviors (viz., detailed roles)

often encountered in components of object-oriented systems (e.g., polling, waiting rendezvous, balking rendezvous, timed rendezvous, random state, etc.). In contrast, we do not try to catalogue typical behaviors of a component but instead conventionalize the transition from scenarios to hierarchical state machines. More precisely, we start with a set of scenarios that express (albeit perhaps partly implicitly) the behavior of their components. We do not assume that the behavior of a component is necessarily typical, that is, that it matches a specific detailed role. Instead, we assemble the behavior of a component from the scenarios in which it is used, leaving room for the combination of detailed roles within a same component, as well as for semantic customization. Furthermore, our approach is iterative: we do not require that all relevant scenarios be known before starting to design the behavior of a component. Instead, new roles can be progressively integrated into the behavior of an existing component.

6 Conclusions

In this paper, we have presented a three-step approach to a systematic transition from scenarios to hierarchical state machines. This approach rests on the definition of inter-scenario relationships (such as containment, alternative, temporal and functional dependency) and of the corresponding families of behavior integration patterns, which we have very briefly introduced but discuss at length elsewhere ([4]).

We are currently working on formalizing scenario temporal and functional dependency. For the former, existing work on temporal logic and notations such as LOTOS [25] serves as a starting point. Our goal is to focus on temporal expressions that have direct impact on state machine design. Similarly, for functional dependency, existing work on causal relationships (in particular in Artificial Intelligence) is scrutinized in order to find forms that bear on state machine design.

Industrial experience with the approach we propose suggests the following benefits:

- Reducing the time required to design complex component behavior
- Increasing the quality of the design of complex component behavior
- Reducing the time required to test complex component behavior
- Reducing undesired scenario interactions
- Increasing traceability between scenarios and detailed component behavior

To conclude, let us remark that the approach described in this paper is constantly evolving as a result of its use in industry, as well as in academia. In particular, our method is followed by the Versatel group of CML Technologies, as well as by our software engineering students at Carleton University. It is also at the basis of a recent university-industry NSERC funding initiative with Nortel Networks to define a standard development process in the context of the Wireless Intelligent Networkss (WIN) standard definition.

References

1. Object Management Group: OMG Unified Modeling Language Specification.
2. Henderson-Sellers, B. and Firesmith, D.: The Open Modeling Language Reference Manual, Cambridge University Press, 1998.

3. Corriveau, J.-P.: Traceability and Process for Large OO projects, IEEE Computer, pp. 63-68, 1996 (also available in a longer version in: Technology of Object-Oriented Languages and Systems (TOOLS-USA-96), Santa-Barbara, August 1996.
4. Bordeleau, F.: A Systematic and Traceable Progression from Scenario Models to Communicating Hierarchical State Machines. Ph.D. Thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1999. (Available at http//www.scs.carleton.ca/~francis)
5. Harel, D.: StateCharts: A Visual Formalism for Complex Systems, Science of Computer Programming, Vol. 8, pp. 231-274, 1987.
6. Buhr, R. J. A. and Casselman R. S.: Use Case Maps for Object-Oriented Systems, Prentice Hall, 1996.
7. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, Addison-Wesley, 1999.
8. Kruchten, P.: The Rational Unified Process: An Introduction, Addison-Wesley, 1999.
9. Bordeleau, F., Corriveau, J.-P., Selic, B.: A Scenario-Based Approach for Hierarchical State Machine Design, Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC'2000), Newport Beach, California, March 15 - 17, 2000.
10. ITU (1996) : Message Sequence Charts (MSC'96), Recommendation Z.120, Geneva.
11. Firesmith, D.: The Pros and Cons of Use Cases, ROAD, May 1996.
12. Jacobson, I. et al.: Object Oriented Software Engineering, Addison-Wesley, 1994.
13. Haugen, O.: MSC Methodology, SISU II Report L-1313-7, Oslo, Norway, 1994.
14. ITU (1993) : Message Sequence Charts (MSC'93). Recommendation Z.120, Geneva.
15. ITU (1992): Specification and Description Language (SDL'92), Recommendation Z.100, Geneva.
16. Koskimies, K., Makinen, E.: Automatic Synthesis of State Machines from Trace Diagrams, Software-Practice and Experience, vol. 24, No. 7, pp. 643-658 (July 1994).
17. Leue, S., Mehrmann, L., Rezai M.: Synthesizing ROOM Models From Message Sequence Charts Specifications, TR98-06, Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada, 1998.
18. ITU (1996) : Message Sequence Charts ('96), Recommendation Z.120, Geneva.
19. Selic, B., Gullickson, G., and Ward, P.T.: Real-time Object-Oriented Modeling, Wiley, 1994.
20. Harel, D., Kugler, H.: Synthesizing State-Based Object Systems from LSC Specifications, Proceedings of Fifth International Conference on Implementation and Application of Automata (CIAA2000), Le
21. tutre Notes in Computer Science, Springer-Verlag, July 2000.
22. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: A System of Patterns, Wiley, 1996.
23. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns-Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.
24. Vlissides, J., Coplien, J.O., Kerth, N.L.: Pattern Languages of Program Design, Addison-Wesley, 1996.
25. Douglass, B.: Doing Hard Time. Addison-Wesley, 2000.
26. ISO LOTOS standard: IS 8807 (http://www.iso.ch/cate/d16258.html)

Towards a Rigorous Semantics of UML Supporting Its Multiview Approach*

Gianna Reggio, Maura Cerioli, and Egidio Astesiano

DISI - Università di Genova (Italy)
email: {reggio,cerioli,astes}@disi.unige.it

Abstract. We discuss the nature of the semantics of the UML. Contrary to the case of most languages, this task is far from trivial. Indeed, not only the UML notation is complex and its informal description is incomplete and ambiguous, but we also have the UML *multiview* aspect to take into account. We propose a general schema of the semantics of the UML, where the different kinds of diagrams within a UML model are given individual semantics and then such semantics are composed to get the semantics on the overall model. Moreover, we fill part of such a schema, by using the algebraic language CASL as a metalanguage to describe the semantics of class diagrams, state machines and the complete UML formal systems.

1 Introduction

UML is the object-oriented notation for modelling software systems recently proposed as a standard by the OMG (Object Management Group), see e.g., [12, 14]. The semantics of the UML, which has been given informally in the original documentation, is a subject of hot debate and of lot of efforts, much encouraged by the OMG itself. Not only users and tool developers badly need a precise semantics, but the outcome of the clarification about semantics is seen of great value for the new version of UML to come.

A most notable effort, which likely preludes to a proposal for semantics to the OMG, is a very recent feasibility study in [3] for an OO meta modelling approach. There are serious reasons why that kind of approach seems the right one for an official semantic definition, if any; in particular the need of seeing UML as an evolving family of evolving languages (that requires a very modular approach) and the request of a notation well-known to the UML users (in this case the meta language would be part of UML).

What we present here is not a proposal for a standard definition and is partly orthogonal and partly complementary to the above direction of work; we even believe, as we will argue later on, that it provides insights and techniques that can be used to fill some scenes in the big fresco of [3].

* Partially supported by Murst - Programma di Ricerca Scientifica di Rilevante Interesse Nazionale Saladin and by CoFI WG, ESPRIT Working Group 29432.

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 171–186, 2001.

Essentially our contribution is twofold:

- we propose a general schema of what a semantics for UML should provide from a logical viewpoint, combining the local view of a single diagram with the general view of an overall UML model;
- we give hints on how to fill some relevant parts of that schema, especially showing how the dynamic aspects fit in.

In this work we adopt rather classical techniques, like modelling processes as labelled transition systems as in CCS et similia, and algebraic specification techniques, supported by a recently proposed family of languages, the `CoFI` family[1] [6]. In particular, since we have to model also the behavioural aspects, we use CASL-LTL [8], which is an extension of CASL, the basic `CoFI` language, especially designed for specifying behaviours and concurrent systems.

We are convinced that explaining the semantics of the UML in terms of well-known techniques helps the understanding of the UML; indeed, as we have shown already in [10], this analysis can easily lead to discover ambiguities and inconsistencies, as well as to highlight peculiarities.

Apart from the work already mentioned [3], several attempts at formalizing the UML have been and are currently under development. Usually they deal with only a part of the UML, with no provision for an integration of the individual diagram semantics toward a formal semantics of the overall UML models, because their final aim is the application of existing techniques and tools to the UML (see [4,13] and the report on a recent workshop on the topic of the UML semantics for the behavioural part [11], also for more references).

In Sect. 2 we will discuss our understanding of the nature of the expected semantics for the UML. In Sect. 3 and 4 we will sketch the semantics of class diagrams and state machines, illustrating our techniques on a running example which is a fragment of an invoice system. Then, in Sect. 5 we will summarize the non-trivial combination of the CASL-LTL specifications representing the semantics of individual diagrams, to get the overall semantics for a UML model.

We want to emphasize that the main contributions of the work presented here are not bound to the particular metalanguage that we used, namely CASL-LTL; what is essential is the overall structure of the semantics and the use of labelled transition systems for modelling behaviour. Still the nice structuring features of CASL-LTL have much facilitated our investigation.

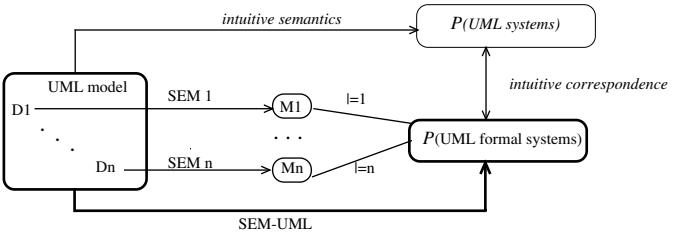
2 A View of the Overall Semantics

A UML model consists of a bunch of diagrams of different kinds, expressing properties on different aspects of a system. In the following we will call UML-*systems* the "real world" systems modeled by using the UML (some instances are information systems, software systems, business organizations). Thus a UML model plays the role of a specification, but in a more pragmatic context.

[1] See the site `http://www.brics.dk/Projects/CoFI`

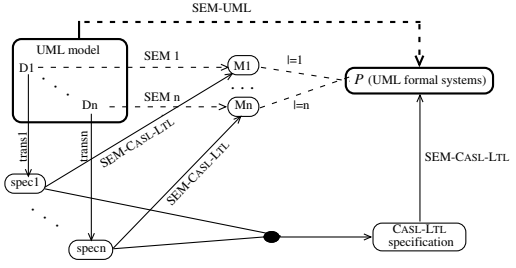
Another analogy that we can establish between UML models and specifications is the fact that the meaning of each diagram (kind) is informally described by [14] in isolation, as well as the semantics of each axiom in logic, and its effect on the description of the overall UML-system is to rule out some elements from the universe of all possible systems (semantic models). Indeed, both in the case of a UML model and of a collection of axioms, each individual part (one diagram or one axiom) describes a point of view of the overall system.

Therefore, our understanding of what should be a general schema for a semantics of the UML is illustrated in the following picture, where the UML *formal systems* are the formal counterparts of the UML-systems.



We have a box representing a UML model, collecting some diagrams of different kinds, and its overall semantics, represented by the arrow labelled by SEM-UML, is a class of UML formal systems. But, each diagram in the model has its own semantics (denoted by the indexed SEM), that is a class of appropriate structures, as well, and these structures are imposing constraints on the overall UML formal systems, represented by lines labelled by the indexed $\models$. A sort of commutativity on the diagram has to hold, that is the overall semantics must be the class of the UML formal systems satisfying all the constraints imposed by the individual semantics. Moreover, the formal semantics must be a rigorous representation of the expected "intuitive semantics", described by the UML standard, [14].

As a technical solution, illustrated in the following picture

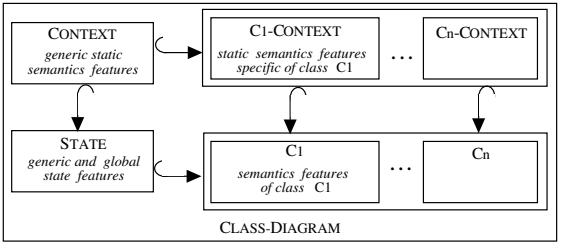


we propose to translate the diagrams (possibly of different kinds) into a common metalanguage, whose formal semantics gives, by composition, the semantics of each diagram in the UML model. Such semantics must correspond to that informally described in [14] and hence the translations are driven by a careful analysis of such document. Moreover, the metalanguage has to provide constructs

to compose the translations of the diagrams in order to get an expression representing the overall UML model. As in the case of the individual diagrams, the (metalanguage) semantics of such expression gives a rigorous semantics to the UML model and hence the composition as well has to be driven by [14]. Because of the need of integrating the formalization of both the static and the dynamic parts of UML, we have found convenient to use as metalanguage an extension of the CASL basic language [6], namely CASL-LTL [8], especially devised to model formally also the dynamic behaviour of the objects.

3 UML Class Diagrams

Let us sketch the algebraic specification describing a given class diagram; its structure is graphically shown below and reflects the logical organization of the information given by the class diagram itself.



Indeed, such specification has four parts.

Two of them, the *generic context part* and the *state part*, are generic, in the sense that they are common to all class diagrams. The *generic context part*, CONTEXT, includes, for instance, the sorts, operations and predicates used to deal with the concepts of object-oriented modeling. The *state part*, STATE, concerns the form of global and (generic) local states.

The other two parts, the *class-context part* and the *class-semantics part* are specific of an individual class diagram and each of them is the sum of smaller specifications, one for each classifier in the class diagram. This structure is chosen in order to reflect as much as possible the structure of the UML model. Analogously to the generic parts, for each classifier C, C-CONTEXT introduces the information about the static semantics (e.g., names of the class, of its attributes and operations), while C introduces information about the semantics (e.g., local states of classes or associations).

Two points worth to keep in mind are that in CASL there is the principle "same-name same-thing" imposing that the realizations of sorts (functions) [predicates] with the same name in different parts of the same overall specification must coincide. Thus, for instance, the semantics of the generic parts that are imported by the specifications representing individual classes must be unique, so that we will have just one global state.

The second point is that the models of a structured specification are not required to be built from models of its subspecifications, that is the structure of the specification is not reflected onto the architecture of the model. Therefore,

the choice about the structuring of the information, for instance by layering the specifications representing individual classes, does not affect the semantics we are proposing for a class diagram, but only its *presentation*.

In the following, we intersperse fragments of the specification with explanations. The overall specifications include the lines proposed here and other analogous parts, that we omit for brevity.

In our specifications, we will use a few standard data types, whose obvious specifications are omitted. Some of them are *parametric*, like for instance LIST[_], where _ is the place holder for the parameter.

3.1 Generic Parts

Context part (Context). First of all, we introduce data types[2] for dealing with UML values and types. The former collect the standard OCL values, like booleans or numbers, and, in particular, the identites of objects, *Ident*. The latter are required in order to translate some OCL constraints and include the names of the standard types as well as the names of classes, *Name*.

sorts $Ident, Name$
preds $isSubType : Name \times Name$
$hasType : Ident \times Name$
axioms $\forall id : Ident; c, c' : Name \bullet isSubType(c, c') \wedge hasType(id, c) \Rightarrow hasType(id, c')$
type $Value ::=$ **sort** $Bool \mid$ **sort** $Date \mid$ **sort** $Integer \mid$ **sort** $Ident \ldots$
$Type ::= bool \mid date \mid integer \mid$ **sort** $Name \ldots$

Other static information have to be recorded as well; for instance we need predicates recording which names[3] correspond to classes (**preds** $isClass : Name$) or associations; attributes/operations (**preds** $isOp : Name \times Name$) of a class; types of the operation input and output (**ops** $argType, resType : Name \to ? List[Type]$); types of the attributes, or of left and right ends of binary associations, etc. Such operations are defined as partial; thus they accept any name as input, but are undefined on those that are not operations/associations. For instance, *argType* will be undefined, if applied to those names not corresponding to an operation. The treatment of other kind of classifiers, e.g., signals, that is analogous, is omitted.

State part (State). Extending the CONTEXT specification, we introduce the constructs to deal with the global states (**sorts** *State*) of the system and the local states of the objects (**sorts** *Object*).

The global states provide information about living instances of the classes, with tools to check if an object exists (**preds** $knownIn : Ident \times State$), to get the local state of an object (**ops** $localState : Ident \times State \to ? Object$), etc.

The local states give information about the identity of the objects and the current values of their attributes. We have operations to get the object identities

[2] The datatype construct in CASL, like **type**$s ::= \ldots$ **sort**$s' \ldots c(\ldots) \ldots$, is a compact notation to declare a sort s stating at the same time that it includes all the elements of s' and that its elements may be as well built by a constructor c.

[3] In the following, we will assume that the names used to represent attributes, operations and classes are all distinct.

(**ops** $getId : Object \to Ident$) and to read (**ops** $rdAttr : Object \times Name \to ? \ Value$) and write (**ops** $wrAttr : Object \times Name \times Value \to ? \ Object$) attributes, as well, with the usual properties about typing (e.g., we cannot assign to an attribute a value of an incompatible type; also updating an attribute is not affecting the others nor the object identity), axiomatically stated.

Each operation of a class represents a nondeterministic (partial) function having as input the operation owner (that is implicit in object-oriented approaches), the global state (in order to use information on the other objects that can be reached by navigation) and the explicit parameter list; the output is the result (if any is expected, else we will use the empty list) and the new global state, possibly modified by some side effect of the operation call. We formalize the operations by the predicate named *call* relating the name of each operation and the inputs to the possible outputs.

preds $call : Ident \times Name \times State \times List[Value] \times List[Value] \times State$

This is one major difference w.r.t. other translations of the UML into logic/algebraic languages, where operations most naturally become functions, that is due to the need of integrating the semantics of class diagrams given in isolation with the sematics of the overall UML model, including concurrent features.

The local states of the associations are represented by a sort; since the case of binary associations is by far the most common in the UML, we are also providing a specialization (**sorts** $BinAssocState < Association$) [4] of the association type for such case, having, for each association end, an operation (**ops** $LAssoc, RAssoc : Ident \times BinAssocState \to Set[Ident]$), yielding the objects that are in relation with a given object that will be used to represent the UML *navigation*.

3.2 Class Specific Parts

To illustrate more concretely how the specification corresponding to a class diagram looks like, we will use as running example a fragment of an invoice system.

We have some passive classes, recording information about clients, products (we do not detail these parts), current (and past) orders and stock of an e-commerce firm, and two active classes, managing such data and representing two kinds of "software" clerks: the *stock handler*, who put the newly arrived products in the stock and removes the correct amount of products to settle an order, and the *invoicer*, who processes orders and sends invoices. Finally, there are associations relating each order respectively to the ordering client and the ordered product. Notice that we had to introduce the stereotype <<external>> in order to annotate that the object kept in the Mailer attribute corresponds to something external to the system.

Class Specific Context Part (C-Context). Each class contributes to the context with the names introduced by the class. So, for instance, the specifica-

[4] Subtyping in CASL is denoted by $<$ and a declaration $s < s'$ implicitly introduces a predicate $\_ \in s$, checking if a term of sort s' is interpreted on a value of type s.

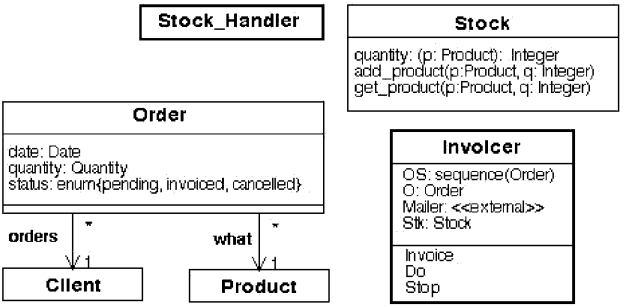


Fig. 1. Class Diagram for the Invoice system

tion for the class `Client` introduces only its name as a constant of sort *Name*, while that for the class `Order` (`Stock`) introduces the names of the attributes (operations) and their typing as well. As the type of some operation of the class `Stock` involves another class, `Product`, we declare the constant with that name, but we do not state that it is a class, as this information is not part of the class `Stock`. When in the end we will put together this specification with that corresponding to the static context of `Product`, we will have the missing information.

spec STOCK-CONTEXT = CONTEXT **then**
ops $Stock, Prod, quantity, addProd, getProd :\rightarrow Name$
axioms
$isClass(Stock)$
$isOp(quantity, Stock) \wedge argType(quantity) = Prod \wedge resType(quantity) = integer...$
We deal with associations in an analogous way with respect to classes.

Class semantics part (C). Each specification corresponding to a class introduces (at least) the sort of the local states of that class objects. Moreover, if we have OCL constraints, then they are translated into axioms.

spec CLIENT = STATE **and** CLIENT-CONTEXT **then**
sorts $Client < Object$
axioms $\forall o : Object; \bullet\ hasType(getId(o), Client) \Leftrightarrow o \in Client$
To translate active classes we use CASL-LTL, an extension of CASL, where sorts may be declared *dynamic*. As we will see in the next section, in this way we have the means to describe the behaviour of their elements. Apart from declaring their object sort as dynamic, active classes are translated like passive ones.

spec STOCKHANDLER0 = STATE **and** STOCKHANDLER-CONTEXT **then**
dsort $StockHandler$ **label** $StockHandler - Label$ **info** $StockHandler - Info$
sorts $StockHandler < Object$
axioms $\forall o : Object; \bullet\ hasType(getId(o), StockHandler) \Leftrightarrow o \in StockHandler$
Also in this part associations are dealt with in an analogous way w.r.t. classes.

The overall specification giving the semantics of the class diagram, called CLASSDIAGRAM, is the union (CASL-LTL keyword **and**) of the specifications representing the individual classes and relations.

4 UML State Machines

In [9] we present our complete formalization of UML state machines associated with active classes using CASL-LTL; a short version has been presented in [10]; here we briefly report the main ideas.

4.1 How to Represent Dynamic Behaviour

Since the handling of the time in the UML, also of the real time, does not require to consider systems evolving in a continuous way, we have to describe a *discrete* sequence of moves, each one of them representing one step of the system evolution. Thus, taking advantage of a well established theory and technology (see, e.g., [5,7]), we model an active object A with a *transition tree*. The nodes in the tree represent the intermediate (interesting) situations of the life of A, and the arcs of the tree the possibilities, or better the *capabilities*, of A of moving from one situation to another. Moreover, each arc is labelled by all the relevant information about the move. In standard transition systems, the label is unstructured; here, for methodological reasons, we prefer to use pairs as decorations, where the first component, called *label*, represents the interaction with the external environment and a second one, called *info*, is some additional information on the move, not concerning interactions.

To describe transition trees, we use *generalized labelled transition systems* (shortly *glts*). A *glts* $(STATE, LABEL, INFO, \rightarrow)$ consists of the sets $STATE$, $LABEL$, and $INFO$, and of the *transition relation* $\rightarrow \subseteq INFO \times STATE \times LABEL \times STATE$. Classical labelled transition systems are the glts where the set $INFO$ has just one element.

Each $(i, s, l, s') \in \rightarrow$ is said to be a *transition* and is usually written $i : s \xrightarrow{l} s'$.

The transition tree for an active object A described by a glts is a tree whose nodes are labelled by states (the root by the initial state of A) and whose arcs are decorated by labels and informations. Moreover, there is an arc decorated by l and i between two nodes decorated respectively by s and s' iff $i : s \xrightarrow{l} s'$. Finally, in a transition tree the order of the branches is not considered, and two identically decorated subtrees with the same root are considered as a unique one.

A glts may be formally specified by using the algebraic specification language CASL-LTL (see [8]), an extension of CASL (see [6]) designed for the specification of processes based on glts's, where we have a construct for the declaration of dynamic sorts:

dsort *State* **label** *Label* **info** *Info*

introducing the sorts *State*, *Label*, and *Info* for the states, the labels and the information of the glts, and implicitly also the transition predicate:

pred $\_ : \_ \xrightarrow{\_} \_ : Info \times State \times Label \times State$

Each model M (an algebra or first-order structure) of a specification including such construct corresponds to the glts $(Info^M, State^M, Label^M, \rightarrow^M)$[5].

[5] Given a Σ algebra A, and a sort s of Σ, s^A denotes the interpretation of s in A; similarly for the operation and predicates of Σ.

In most cases we will be interested in a *minimal* glts, where the only transitions are those explicitly required by the axioms. This is, for instance, the case when translating active classes, because their activity is required to be fully described by the corresponding state machine. To achieve this result we restrict the specification models to the (minimal) *initial* ones[6] (CASL keyword **free**).

Active classes as glts. Assume to have a given active class ACL with a given associated state machine SM. For instance, let us consider the following state machine, associated with the active class Invoicer of our running example. The

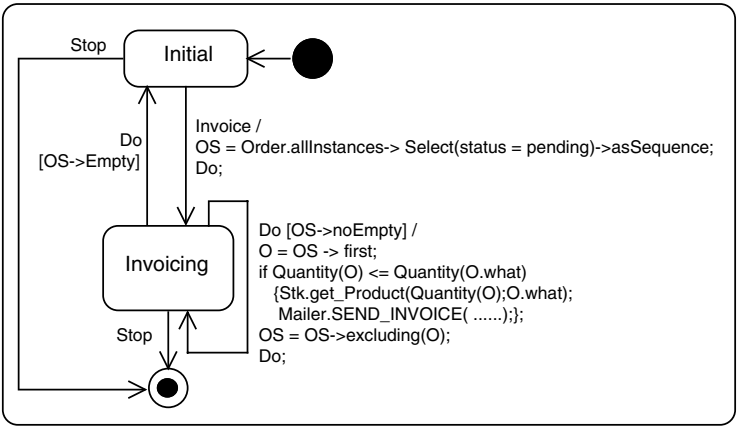


Fig. 2. The State Machine Associated with Invoicer

instances of ACL, called *active objects* in the UML terminology, are just processes and we model them by using a glts. We algebraically specify such glts, named in the following *GLTS*, with the specification ACL-DYNAMIC.

It is important to notice that the states and the transitions of the state machine SM are different in nature from, and are not in a bijective correspondence to, those of the corresponding *GLTS*. Indeed, one SM-transition corresponds to a sequence of *GLTS*-transitions and the *GLTS*-states are more detailed than those of the SM. To clarify the relationship, see a small fragment of the transition tree associated with the state machine for the Invoicer class (to simplify the picture we only report the configuration and the event queue of each *GLTS*-state). Each *GLTS*-transition corresponds to performing a part of a state machine transition. Then the atomicity of the transitions of SM (run-to-completion assumption) required by [14] will be guaranteed by the fact that, while executing the various parts of a transition triggered by an event, the involved threads cannot dispatch other events.

[6] If any, otherwise the model set becomes empty.

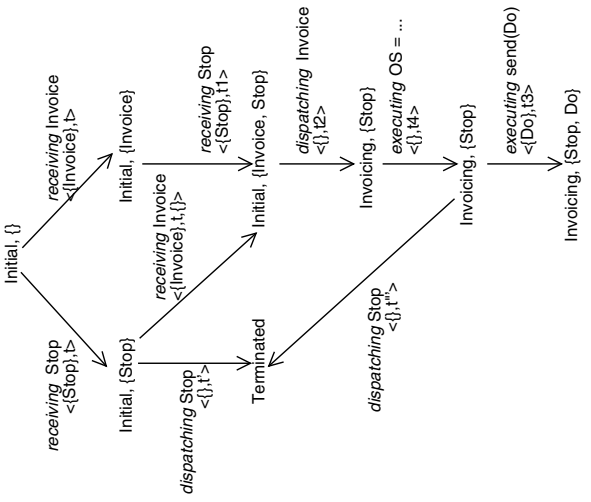


Fig. 3. A fragment of a transition tree

We translate SM into the following specification ACL-DYNAMIC:

spec ACL-DYNAMIC =
LABEL **and** ACL-STATE **and** INFO **then**
free {
dsort *Acl* **label** *Label* **info** *Info*
axioms
. . . . . .
} end

Following the UML philosophy, we assume that all "static" information about the class ACL (and the others), like for instance the attributes and operations, are found in the class diagram and hence in the specification CONTEXT, defined in Sect. 3.

As in the case of class diagrams, a large part of the specification of the *GLTS* representing a state machine is generic, that is, it is common to all state machines. Actually, the only point where the features of an individual state machine play a role is in the definition of the function associating each set of states of the state machine with the set of the transitions starting from it, discussed at the end of the section.

To illustrate the modularity of the approach, we report the axiom defining the transitions corresponding to dispatch an event; for the others see [9].

The intuitive meaning of the axiom is that, if

- an active object is not fully frozen executing run-to-completion steps,
- an event *ev* may be dispatched,
- *rds* is a set of correct interactions for the class ACL corresponding to read the attributes of other objects (checked by *Ok_Read_*ACL),

then *GLTS* has a transition,

- with the label made by *rds* and the received time t,
- where *ev* has been dispatched and the history has been extended with the current states.

$$not\_frozen(conf) \ \wedge \ dispatchable(ev, e\_queue) \ \wedge \ Ok\_Read\_\mathsf{ACL}(rds) \ \wedge$$
$$\mathsf{Dispatch}(ev, conf, e\_queue, rds, attrs, id) = conf', e\_queue' \ \Rightarrow$$
$$id : \langle conf, attrs, e\_queue, history, chinf \rangle \xrightarrow{\langle rds, t \rangle}$$
$$id : \langle attrs, conf', e\_queue', history', chinf \rangle$$

where $history' = \langle active\_states(conf), t \rangle \ \& \ history$

In such axiom we use the auxiliary operation Dispatch: $\mathsf{Dispatch}(ev, conf, e\_queue, rds, attrs, id) = conf', e\_queue'$ holds whenever the object *id* with configuration *conf* and event queue *e_queue* using *rds* and *attrs* may dispatch the event *ev* changing its configuration to *conf′* and its event queue to *e_queue′*.

Notice that the axiom is independent from the particular state machine to be translated. However, the (omitted) specification of the function Dispatch is based on the function **Trans** associating with each set of states (of the state machine) S the set of the transitions having S as source, and its specification is different for each state machine. For instance, **Trans** for the state machine for the Invoicer class associates two transitions to $\{Initial\}$, three to $\{Invoicing\}$ and none to the final state.

5 Semantics of UML Models

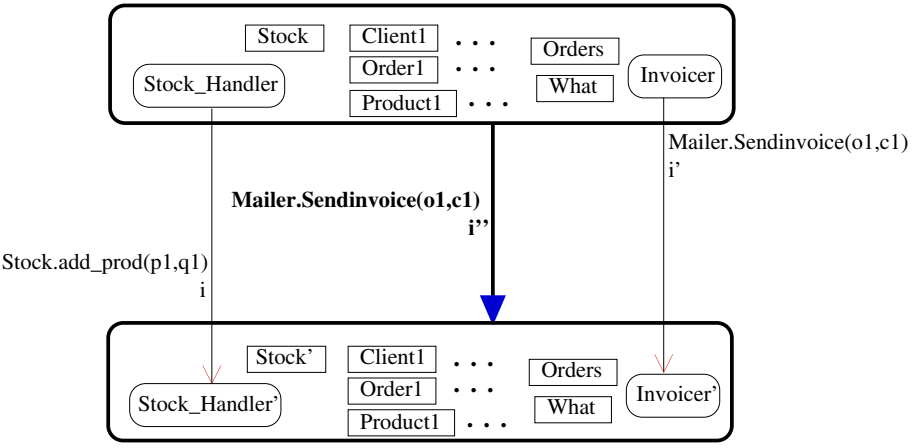
The overall semantics of a UML model is given by a set of what we have called (see Sect. 2) UML formal systems. First we briefly illustrate them also with the help of our example and then we outline their formal specification.

5.1 UML Formal Systems

A UML model describes the *structure* of a system, that is which are the components and which are their capabilities, and the *behaviour* of the system itself, that is the evolution of its components along the time and the interactions of the system with the external world (e.g., with the users).

Therefore, its semantics is a set of acceptable structured processes that we represent again by means of *GLTS*, and that we call UML *formal systems*. But, since reducing the concurrency among the components of a system to interleaving appears to be reductive, we want to use *structured GLTS*, where the transitions correspond to really concurrent executions of sets of transitions of the system subcomponents. The sorts *State*, *Info* and *Label* of a *GLTS* can be endowed with operations (and predicates) and be defined, or built as datatypes starting from other (dynamic) sorts. Thus, we can easily impose a state of the structured *GLTS* to be a set of states of its subcomponents and its transitions and labels to be defined in terms of those of its active parts.

For instance, a possible transition of a UML formal system corresponding to our running example is the following:



Both in the source and in the target state we have two active components (represented by rounded shapes), the Stock_Handler and the Invoicer and a number of passive components (represented by rectangles): some instances of Client and Product, the unique instance of Stock and the state of the associations orders and what. In this picture we consider the parallel execution of two activities, graphically represented by the thin lines connecting the involved active components of the source and target state:

- the Stock_Handler adds to the Stock some quantity *q1* of the product *p1*; the effect of this action is to change the state both of the Stock_Handler and of the Stock;
- the Invoicer sends to the client *c1* an invoice for the order *o1*, by calling a method of some external *mailer*; thus, the effect of this action is to change the state of the Invoicer and to communicate with the external world through the label of the transition.

The parallel composition of these actions is described by a transition, graphically represented by the thick arrow connecting source and target structured states. Notice that, since the labels of the structured system carry only information about the interactions with the external world, the label of this transition is taking into account only the call to the mailer. But the resulting state of the system is modified because the internal states of all the objects involved in the move are (possibly) modified.

5.2 The Specification of a UML Formal System

The semantics of a whole UML model is given by the combination of the specifications providing the semantics of the single views determined by the various diagrams that compose the model. Therefore, we start by extending the specifications produced by the class diagram(s) and the state machines in order to describe the glts modeling the system.

The states of the overall system are sets of states of the components. Indeed, we do not need multisets, since objects are distinguished by their identities.

type *State* ::= Λ | **sort** *Object* | **sort** *Association* | $\_||\_(State; State)$
op $\_||\_ : State \times State \to State$ **assoc**, **comm**, **idem**, **unit**: Λ
The specification of the labels of the system (System-Label), based on the interactions with the external environment, is omitted. The informations of the generalized transitions will be a set of stimuli. Indeed, to be able to evaluate the satisfaction of a sequence diagram by a UML formal system M we need to record for each transition of M which are the stimuli exchanged during such transition. The specification Stimulus, describing the stimuli, that is the the trivial translation into Casl of the UML stimuli (see [14] p. 2-86) is omitted as well.

In a context extending the specifications Class_Diagram, Acl-Dynamic$_1$, ..., Acl-Dynamic$_p$, System-Label and FiniteSet[Stimulus], we can describe the dynamics of the elements of sort *State*
dsort *State* **label** *System-Label* **info** *FinSet*[*Stimulus*]

It is worth noting that to state the behavioural axioms we need some temporal logic combinators available in Casl-Ltl that we have no space to illustrate here. The expressive power of such temporal logic will be also crucial for the translation of sequence diagrams, though they are not discussed here.

Though most of the identifications needed are already given by the "same-name same-thing" principle of Casl, we also need axioms stating that an identity is known in a system state iff it corresponds to a component of the system, that can be easily inductively defined, or axioms stating that for active classes getting the identity corresponds to selecting the identity component of their state tuples. There are several such axioms, that, though quite trivial to be expressed, would produce a long boring list.

Much more interesting are the axioms describing the transitions of the overall system. For instance, we can state that we can add to the source and the target of a transition any number of components that do not participate into the transition: **axiom** $inf : s \xrightarrow{l} s' \Rightarrow inf : s||s_0 \xrightarrow{l} s'||s_0$.

Let us now show how to compose the transitions of the individual active components, using several auxiliary functions, whose axiomatization we omit for lack of room. Assume that we have, as part of a state s, a group of active objects a_i which may perform some transitions $inf_i : a_i \xrightarrow{l_i} a_i'$, accordingly to the specification of their active classes. Then, we can compose such transitions into a transition of s only if the interactions appearing on the l_i's, and on the label l of the resulting transition are pairwise matched, i.e., any thing sent is received by the addressee, including the external environment, and only sent things are received (this is checked by the predicate *Ok_Labels*) and the interactions corresponding to read the attributes of passive objects are in agreement with their actual values (this is checked by the predicate *Read_Attributes*). If these conditions are satisfied, then the system can move into a new state where the active objects are evolved into the a_i''s, the passive components of s are updated and some new components may be possibly added.

$$\wedge_{i=1}^{n}\ inf_i : a_i \xrightarrow{l_i} a_i' \wedge$$
$$Ok\_Labels(l_1 \dots l_n l) \wedge$$
$$Read\_Attributes(l_1 \dots l_n l, pss) \wedge$$
$$Updated(l_1 \dots l_n l, pss) = pss' \wedge$$
$$Created(l_1 \dots l_n l) = \overline{oss} \Rightarrow$$
$$Stimuli\_Of(l_1 \dots l_n l, inf_1, \dots, inf_n) : a_1 || \dots || a_n || pss \xrightarrow{l} a_1' || \dots || a_n' || pss' || \overline{oss}$$

It is interesting to note that there is no guarantee that the specification of the overall system is consistent. Indeed, if, for instance, the constraints imposed by the class diagrams are not met by the behaviour described by the state machines, then the UML model corresponds to no systems and this is shown by the fact that the overall specification is inconsistent (i.e., it has no models).

6 Conclusion

Our work stems from the belief, supported by concrete experience, that analysing the UML with different techniques helps the understanding, especially when an official formal definition of its semantics is lacking and many proposal for improvements and extensions are under way. Indeed, this kind of analysis is explicitly encouraged by some members of the OMG involved in the definition of UML (Bran Selic during the discussion at the <<UML>> 2000 Workshop "Dynamic Behaviour in UML Models: Semantic Questions", York, October 2000). Moreover, we have already shown in [10] how this kind of analysis can lead to spot problematic points and offer various alternatives for a precise definition.

With respect to the proposal of a standard definition by an OO metamodelling approach, like the one advocated by [3], our work is complementary, in two senses. First of all, it provides a basic technique for modelling dynamic behaviour, which is based on the twenty years long and successful experience of CCS and the like (labelled transition systems), already used for the full formal definition of Ada ([1]). That technique is adequate for modelling all dynamic features of UML; for example by this technique we are currently working on sequence diagrams; some other kinds of diagrams that we have partly analyzed, and that we conjecture can be added to our schema without major problems, are

- the collaboration diagrams, being rather similar to the sequence diagrams;
- the activity diagrams, as they are a specialization of the statechart diagrams.

Thus it would be interesting to explore the possibility of adopting this technique as a basis in the so called dynamic-core sketched in [3], which is admittedly rather preliminary. For a proposal in that direction see [2]. A point that we want to stress again is that our approach is supporting the multiview philosophy, in the sense that each part of a UML model has a proper formalization that is integrated with those of the other parts. Moreover, we are considering full UML and this is quite important, because it is often the case that the semantics given for a restricted part of UML in isolation is useless when trying to uplift it to the full UML.

For instance, even from this partial work concerning just two kinds of diagrams, it results clear that the separation of concerns apparently achieved by using class diagrams to describe the system structure and state machines to capture the system dynamics is limited. Indeed, the class diagram imposes restrictions on the dynamic behaviour of the objects, through the constraints on the operations and on the classes. Vice versa, a state machine cannot be considered in isolation, as we need to know whether it is associated with an active or with a passive class and which are the operations/signals/attributes of such class and of the other classes. Moreover, we have also found that we need to know how a UML model interacts with its external environment, in order to describe the labels of the overall system (see Sect. 5.2).

We expect that furthering our investigation to other kinds of diagrams more relationships among the parts of a UML model will be exposed, deepening our understanding of the UML and paving the way for better new versions and a standard complete definition of its semantics.

References

1. E. Astesiano, C. Bendix Nielsen, N. Botta, A. Fantechi, A. Giovini, P. Inverardi, E. Karlsen, F. Mazzanti, J. Storbank Pedersen, G. Reggio, and E. Zucca. The Draft Formal Definition of Ada. Deliverable, CEC MAP project: The Draft Formal Definition of ANSI/STD 1815A Ada, 1986.
2. E. Astesiano and G. Reggio. A Proposal of a Dynamic Core for UML Metamodelling with MML. Technical Report DISI-TR-01-1, DISI – Università di Genova, Italy, 2001.
   `ftp://ftp.disi.unige.it/person/ReggioG/AstesianoReggio01a.pdf`.
3. T. Clark, A. Evans, S. Kent, S. Brodsky, and S. Cook. A Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modeling Approach - Version 1.0. (September 2000). Available at
   `http://www.cs.york.ac.uk/puml/mmf.pdf`., 2000.
4. R. France and B. Rumpe, editors. *<<UML>>'99 - The Unified Modelling Language*. Number 1723 in Lecture Notes in Conputer Science. Springer Verlag, 1999.
5. R. Milner. *Communication and Concurrency*. Prentice Hall, London, 1989.
6. The CoFI Task Group on Language Design. CASL Summary. Version 1.0. Technical report, 1998. Available on
   `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/`.
7. G. Plotkin. An Operational Semantics for CSP. In D. Bjorner, editor, *Proc. IFIP TC 2-Working conference: Formal description of programming concepts*. North-Holland, Amsterdam, 1983.
8. G. Reggio, E. Astesiano, and C. Choppy. CASL-LTL : A CASL Extension for Dynamic Reactive Systems – Summary. Technical Report DISI-TR-99-34, DISI – Università di Genova, Italy, 1999.
   `ftp://ftp.disi.unige.it/person/ReggioG/ReggioEtAll99a.ps`.
9. G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. A CASL Formal Definition of UML Active Classes and Associated State Machines. Technical Report DISI-TR-99-16, DISI – Università di Genova, Italy, 1999. Revised March 2000. Available at `ftp://ftp.disi.unige.it/person/ReggioG/Reggio99b.ps`.

10. G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach. In T. Maibaum, editor, *Proc. FASE 2000 - Fundamental Approaches to Software Engineering*, number 1783 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2000.
11. G. Reggio, A. Knapp, B. Rumpe, B. Selic, and R. Wieringa (editors). Dynamic Behaviour in UML Models: Semantic Questions. Technical report, Ludwig-Maximilian University, Munich (Germany), 2000.
12. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1999.
13. S.Kent, A.Evans, and B. Rumpe. UML Semantics FAQ . In A. Moreira and S. Demeyer, editors, *ECOOP'99 Workshop Reader*. Springer Verlag, Berlin, 1999.
14. UML Revision Task Force. *OMG UML Specification*, 1999. Available at `http://uml.shl.com`.

Towards Development of Secure Systems Using UMLsec

Jan Jürjens*

Computing Laboratory, University of Oxford, GB

Abstract. We show how UML (the industry standard in object-oriented modelling) can be used to express security requirements during system development. Using the extension mechanisms provided by UML, we incorporate standard concepts from formal methods regarding multi-level secure systems and security protocols. These definitions evaluate diagrams of various kinds and indicate possible vulnerabilities.
On the theoretical side, this work exemplifies use of the extension mechanisms of UML and of a (simplified) formal semantics for it. A more practical aim is to enable developers (that may not be security specialists) to make use of established knowledge on security engineering through the means of a widely used notation.

1 Introduction

There is presently an increased need to consider security aspects when developing systems (that may e.g. communicate over untrusted networks). This is not always met by adequate knowledge on the side of the developer. This is problematic since in practice, security is compromised most often not by breaking the dedicated mechanisms (such as encryption or security protocols), but by exploiting weaknesses in the way they are being used [And94]. Thus security mechanisms cannot be "blindly" inserted into a security-critical system, but the overall system development must take security aspects into account.

Object-oriented systems offer a very suitable framework for considering security due to their encapsulation and modularisation principles.

The Unified Modeling Language (UML) [RJB99] is an industry standard for specifying object-oriented software systems. Compared to other modelling languages, UML is very precisely defined.

The aim of this work is to use UML to encapsulate knowledge on prudent security engineering and thereby make it available to developers not specialised in security.

This work profits from a work towards a formal semantics for UML e. g. in [EFLR99,RACH00]. Its aim is to make formalism have an impact in the actual software development process, [Hußmann,AR00].

* `http://www.jurjens.de/jan` – `jan@comlab.ox.ac.uk` - Supported by the Studienstiftung des deutschen Volkes and the Computing Laboratory.

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 187–200, 2001.

For space limitations, we present our results in the framework of a simplified fragment of UML (in its current version 1.3 [For99]), the extension to the whole of the actual UML is under development [Jür01b].

After presenting some background and related work in the following subsection, we summarise our use of UML in the next section. In the later sections we show how to use four central kinds of diagrams of UML to develop security-critical systems. We end with a conclusion and indicate future work.

1.1 Background and Related Work

A traditional way of ensuring security in computer systems is to design *multi-level secure* systems (LaPadula, Bell 1973). There are levels of sensitivity of data (usually *high* and *low*). To ensure confidentiality of data, one enforces the policy that information always flows up: *low* data may influence *high* data, but not vc. vs. (the opposite of this condition provides data *integrity*; thus also integrity is implicitly covered by our approach). To avoid *covert channels* one can use the notion of *noninterference* (Goguen, Meseguer 1982; cf. e. g. [Jür00]). Secure communication over untrusted networks requires specific mechanisms such as encryption and cryptographic protocols.

There has been much work on security using formal methods (e. g. [BAN89, RWW94,Low96,AJ00,Jür01c]), mostly about security protocols or secure information flow.

[And94] suggests to use software engineering techniques to ensure security. [DS00] proposes to "extend the syntax and semantics of standards such as UML to address security concerns". We are not aware of any published work towards this goal.

Our work depends on concepts from the considerable work towards providing a formal semantics for UML (e. g. [BGH+98,EFLR99,GPP98,RCA00,RACH00, BD00,Öve00]). In particular the security notions concerning behaviour cannot be formulated at the level of abstract syntax, requiring a formal semantics to express and reason about the security requirements. As a formal semantics for UML is subject of ongoing research, we use a (simplified) semantics tailored to our needs for the time being. It would be preferable to employ a universally defined formal semantics, therefore our work may contribute to exemplify the need for a formal semantics for UML.

2 Developing Secure Systems with UML

UML consists of diagram types describing views on a system (an excellent introduction is [SP00]). We use an extension of UML (called UMLsec) to specify standard security requirements on security-critical systems. Our results may be used with any process; security-critical systems usually require an iterative approach (e. g. the "Spiral") [And94].

We concentrate on the most important kinds of diagrams for describing object-oriented software systems (following [RACH00]):

Class diagrams define the static structure of the system: classes with attributes and operations/signals and relationships between classes. We use them to ensure that exchange of data obeys security levels.

Statechart diagrams give the dynamic behaviour of an individual object: events may cause state in change or actions. We use them to prevent indirect information flow from high to low values within an object.

Interaction diagrams describe interaction between objects via message exchange. We use sequence diagrams to ensure correctness of security-critical interaction between objects (especially in distributed object systems).

Since security of a software system depends on the security of the underlying physical layer, we additionally use **deployment diagrams** to ensure that security requirements on communication are met by the physical layer.

To specify security levels we use the UML extension construct of a *tag* to indicate model elements with high security level.

We formulate our concepts at this point on the level of abstract syntax as far as possible (i. e. those concerning static aspects), given in set-theoretical terms for brevity. Behavioural aspects (modeled using statechart and sequence diagrams) can not be treated this way. In absence of a general formal semantics we use a specifically defined one which covers just the aspects needed and to the needed degree of detail, to follow space restrictions (a more complete account is to be found in [Jür01b]).

We stress that the aspects that are left out (such as association and generalisation in the case of class diagrams) can and should be used in the context of our usage of UML; they do not appear in our presentation simply because they are not needed to specify the considered properties (and for lack of space).

It should also be stressed that the formal semantics is used here only to convey our ideas on how to model security aspects of systems using UML and not to provide a semantics for general use. We will translate our definitions to a generally defined formal semantics when available.

3 Classes and State

We give the abstract syntax for class models. Besides the usual information on attributes and operations/signals we also have interfaces. Additionally, the tag {high} is used to mark data items (attributes, arguments of operations or signals or return values of operations) of the corresponding security level (absence of this tag is interpreted as the level low).[1] In the concrete syntax, these tags are written behind the data types. Following [RACH00] we assume that the attributes are fully encapsulated by the operations, i. e. an attribute of a class can be read and updated only by the class operations.

[1] More precisely, UML provides tag-value pairs. Here we use the convention that where the values are supposed to be boolean values, they need not be written (then presence of the label denotes the value true, and absence denotes false).

For brevity, our abstract syntax only gives the aspects we use to specify our security condition. [EFLR99] has a more complete account.

An *attribute specification* $A = (\mathsf{att\_name}, \mathsf{att\_type}, \mathsf{init\_value}, \mathsf{att\_tags})$ is given by a name att_name, a type att_tags, an initial value init_value[2] and a set of tags att_tags (here we only care if it contains the tag {high} or not, determining the security level of the attribute).

An *operation specification* $O = (\mathsf{op\_name}, \mathsf{Arguments}, \mathsf{op\_type}, \mathsf{re\_tags})$ is given by a name op_name, a set of Arguments, the type op_type of the return value and a set re_tags of tags denoting the security level of the return value. The set of arguments may be empty and the return type may be the empty type $\emptyset$ denoting absence of a return value. An *argument* $A = (\mathsf{arg\_name}, \mathsf{arg\_type}, \mathsf{arg\_tags})$ is given by its name arg_name, its type arg_type and a set arg_tags denoting the security level of the argument.

A *signal specification* is like an operation specification, except that there is no return type and no corresponding set of tags.

An *interface* $I = (\mathsf{int\_name}, \mathsf{Operations}, \mathsf{Signals})$ is given by a name int_name and sets of operation names Operations and signal names Signals giving the operations and signals that can be called or sent through it.

A *class model* $C = (\mathsf{class\_name}, \mathsf{AttSpecs}, \mathsf{OpSpecs}, \mathsf{SigSpecs}, \mathsf{Interfaces}, \mathsf{State})$ is given by a name class_name, a set of attribute specifications AttSpecs, a set of operation specifications OpSpecs, a set of signal specifications SigSpecs, a set of class interfaces Interfaces and a statechart diagram State giving the object behaviour. We require that in the set of attribute specifications, the attribute names are mutually distinct, so that an attribute is uniquely specified by its name and the name of its class (and similarly for operation and signal specifications and class interfaces).

3.1 Statechart Diagrams

We fix a set Var of (typed) variables $x, z, y, \ldots$ used in statechart diagrams. At this point, we only consider simple states with one thread of execution, because the formal semantics of more complex features raises some questions [RACH00].

We define the notion of a statechart diagram for a given class model C: A *statechart diagram* $S = (\mathsf{States}, \mathsf{init\_state}, \mathsf{Transitions})$ is given by a set of States (that includes the initial state init_state) and a set of Transitions. (On the level of concrete syntax, the initial state is the state with an in-going transition from the start marker.)

A *statechart transition* $t = (\mathsf{source}, \mathsf{event}, \mathsf{guard}, \mathsf{Actions}, \mathsf{target})$ has a source state, an event, a guard, a list of Actions and a target state. Here an *event* is the name of an operation or signal with a list of distinct variables as arguments that is assumed to be well-typed (e. g. $\mathsf{op}(x, y, z)$). Let the set Assignments consist of all partial functions that assign to each variable and each attribute of the class C a value of its type (partiality arises from the fact that variables may be undefined). A *guard* is a function $g : \mathsf{Assigments} \to \mathsf{Bool}$ evaluating each

[2] We assume that these are specified in the class rather than by the creating object.

assignment to a boolean value. (On the level of UML's concrete syntax, it is left open how to write such guards; often it is done in OCL.) An *action* can be either to assign a value v to an attribute a (written $a := v$), to call an operation op resp. to send a signal sig with values $v_1, \ldots, v_n$ (written $\mathsf{op}(v_1, \ldots, v_n)$ resp. $\mathsf{sig}(v_1, \ldots, v_n)$), or to return values $v_1, \ldots, v_n$ as a response to an earlier call of the operation op (written $\mathsf{return}_{\mathsf{op}}(v_1, \ldots, v_n)$). In each case, the values can be constants, variables or attributes (and need to be well-typed). In the case of *output actions* (calling an operation or sending a signal) we include the types of the arguments (and possibly of the return value) together with any security tags. (This is not necessary in the usual UML syntax; we need this extra information later.)

Interpreting statechart diagrams. To define our security condition on statechart diagrams we need to interpret them as state machines. Due to space constraints we can only sketch this interpretation, a detailed account is in [Jür01b].

Suppose we are given a class model C with a statechart diagram S. We define the associated state machine $M_S \stackrel{\text{def}}{=} (\mathcal{S}, i, \mathcal{T})$ to consist of

- a set of states $\mathcal{S} \stackrel{\text{def}}{=} \mathsf{States} \times \mathsf{Assignments}$,
- an initial state $i \stackrel{\text{def}}{=} (\mathsf{init\_state}, \mathsf{init\_as})$ (where the initial assignment $\mathsf{init\_as}$ maps each attribute to its initial value and is undefined on the set of variables) and
- a set $\mathcal{T}$ of state machine transitions $t = (s, e, A, s')$ (for states s, s', a trigger e and a list of actions A) defined as follows. A trigger is the name of an operation or signal with a list of (suitably typed) values as arguments.

Firstly, for any assignment function ϕ, a list of variable or attribute names $\boldsymbol{x} = (x_1, \ldots, x_n)$ and a list of values $\boldsymbol{v} = (v_1, \ldots, v_n)$ of the corresponding types, we define $\phi[\boldsymbol{x} \mapsto \boldsymbol{v}]$ by $\phi[\boldsymbol{x} \mapsto \boldsymbol{v}](z) \stackrel{\text{def}}{=} \phi(z)$ for an attribute or variable z not in $\boldsymbol{x}$ and $\phi[\boldsymbol{x} \mapsto \boldsymbol{v}](x_i) \stackrel{\text{def}}{=} v_i$ for $i = 1, \ldots, n$.

Suppose we are given a transition $t = (\mathsf{source}, \mathsf{event}, \mathsf{guard}, \mathsf{Actions}, \mathsf{target}) \in \mathsf{Transitions}$ in the statechart diagram (where the event has the list of variables $\boldsymbol{x} = (x_1, \ldots, x_n)$), a corresponding list of values $\boldsymbol{v} = (v_1, \ldots, v_n)$ and an assignment ϕ such that $\mathsf{guard}(\phi[\boldsymbol{x} \mapsto \boldsymbol{v}]) = \mathsf{true}$. Denote the list of attributes that are assigned new values in the list of $\mathsf{Actions}$ by $\boldsymbol{a}$ and the corresponding list of values by $\boldsymbol{w}$. We define the state machine transition $t_{\boldsymbol{v},\phi} \stackrel{\text{def}}{=} ((\mathsf{source}, \phi), \mathsf{event}(\boldsymbol{v}), \mathsf{Actions}', (\mathsf{target}, \phi[\boldsymbol{x} \mapsto \boldsymbol{v}][\boldsymbol{a} \mapsto \boldsymbol{w}]))$ where $\mathsf{event}(\boldsymbol{v})$ is the trigger obtained from event by substituting the values $\boldsymbol{v}$ for the variables $\boldsymbol{x}$ and $\mathsf{Actions}'$ is the list of actions obtained from $\mathsf{Actions}$ by removing the attribute assignments and by instantiating any variables or attributes appearing as parameters using the assignment $(\phi[\boldsymbol{x} \mapsto \boldsymbol{v}])[\boldsymbol{a} \mapsto \boldsymbol{w}]$. Then we define $\mathcal{T} \stackrel{\text{def}}{=} \{t_{\boldsymbol{v},\phi} : \mathsf{guard}(\phi[\boldsymbol{x} \mapsto \boldsymbol{v}]) = \mathsf{true}\}$.

Assume for the moment that the state machine under consideration is deterministic in the sense that at any state, any given trigger can fire at most one transition. Any state s of the state machine defines a function $[s]$ from sequences

of triggers to sequences of actions as follows. For a given sequence of triggers $\boldsymbol{e} = (e_1, \ldots, e_n)$ we obtain the actions performed by an object in reaction to these triggers as follows: If there exists a transition $s \stackrel{e_1;a_1,\ldots,a_m}{\longrightarrow} s'$ for some list of actions $a_1, \ldots, a_m$ (which is the only transition triggered by e_1 by determinism), then define $[s](\boldsymbol{e}) \stackrel{\text{def}}{=} (a_1, \ldots, a_m).[\![s]\!]((e_2, \ldots, e_n))$ (where . denotes concatenation of tuples), otherwise define $[s](\boldsymbol{e}) \stackrel{\text{def}}{=} [\![s]\!]((e_2, \ldots, e_n))$ (here we follow the UML convention that events with unspecified reaction are ignored).

Thus any deterministic state machine S gives a function $[\![S]\!]$ from sequences of triggers to sequences of actions per $[\![S]\!](\boldsymbol{e}) \stackrel{\text{def}}{=} [\mathsf{init\_state}_S](\boldsymbol{e})$. We generalise this to *nondeterministic* state machines S by defining $[\![S]\!]$ to be the set of functions $[\![T]\!]$ for all (deterministic) state machines T constructed by choosing one transition for each trigger in each state in S (where such a transition exists).

Note that we follow [RACH00] in considering sequences of events rather than multisets (*queues* in UML terminology) to ensure preservation of event orderings *within* a state machine, but we can easily derive the case for multisets by considering all possible sequences that give rise to a given multiset. – Note that in the *interaction* between state machines, preservation of message ordering can not be assumed.

We define the *low view* $\mathcal{L}(\boldsymbol{e})$ of a sequence $\boldsymbol{e}$ of triggers (resp. actions) to be the sequence obtained from $\boldsymbol{e}$ by substituting all arguments of types marked $\{\mathsf{high}\}$ by the symbol $\Box$.

For example, the low view of the action sequence $(\mathsf{op}_1(2, 5, 7), \mathsf{op}_2(4, 3))$, with operation specifications

$$(\mathsf{op}_1, ((a_1, \mathsf{int}, \emptyset), (a_2, \mathsf{int}, \mathsf{high}), (a_3, \mathsf{int}, \emptyset)), \mathsf{int}, \{\mathsf{high}\}) \text{ and}$$
$$(\mathsf{op}_2, ((a_3, \mathsf{int}, \{\mathsf{high}\}), (a_4, \mathsf{int}, \{\mathsf{high}\})), \mathsf{int}, \emptyset)$$

is $(\mathsf{op}_1(2, \Box, 7), \mathsf{op}_2(\Box, \Box))$.

Definition 1 *An object* preserves security *if in its statechart diagram S, no* low *output value depend on* $\{\mathsf{high}\}$ *input values, i. e. if for all functions $h \in [\![M_S]\!]$ (where M_S is the associated state machine) and all trigger sequences $\boldsymbol{e}, \boldsymbol{f}$, $\mathcal{L}(\boldsymbol{e}) = \mathcal{L}(\boldsymbol{f})$ implies $\mathcal{L}(h(\boldsymbol{e})) = \mathcal{L}(h(\boldsymbol{f}))$.*

This is a simple generalisation of the notion of *noninterference* [GM82] to the nondeterministic case. It is possible to refine this notion to allow *encrypted* low data to depend on high data (e. g. to encrypt high data and then send it out on an untrusted network); we cannot give the details here for lack of space.

Example: Entry in multi-level database. The object in Figure 1 does not preserve security (the operation $\mathsf{rx}()$ leaks information on the account balance).

3.2 Class Diagrams

A *class diagram* $D = (\mathsf{Cls}, \mathsf{Dependencies})$ is given by a set Cls of class models and a set of $\mathsf{Dependencies}$. A *dependency* is a tuple $(\mathsf{client}, \mathsf{supplier},$

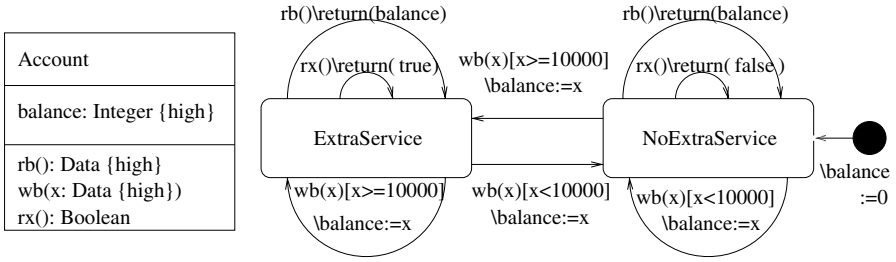


Fig. 1. Multi-level database

interface, stereotype) consisting of class names client and supplier (signifying that client depends on supplier), an interface name interface (giving the interface of the class supplier through which client accesses supplier; if the access is direct this field contains the client name) and a stereotype which for our present purposes can be either «send» or «call». We require that the names of the class models are mutually distinct.

Definition 2 *A class diagram D gives* secure dependency *if whenever there is a dependency with stereotype* «send» *or* «call» *from a client A to an interface I of a supplier B then for each operation or signal o specified in I the following holds:*

- *the security levels on the argument values of o in the statechart diagram of A agree with those of o in the class diagram of B and*
- *the security levels of the return value of o (when o is an operation) in the class diagram of B agree with those of o in the statechart diagram of A.*

If the statechart diagram is not specified for a certain class model, the security levels can be given in a special model element newly introduced here for this purpose, the *output action list*, which specifies the operations an object can call and the signals an object can send.[3] An output action list is a classifier given by a rectangle with the keyword «output actions», carrying the name of the corresponding class and with the operations and signals (with their types and security levels) listed in a compartment of the rectangle. It is similar but dual to an interface.

Example. In Figure 2, the random() operation of the server does not provide the security level required by the client for the seed.

[3] In future work we will examine the possibility to employ UML-RT capsules in this context.

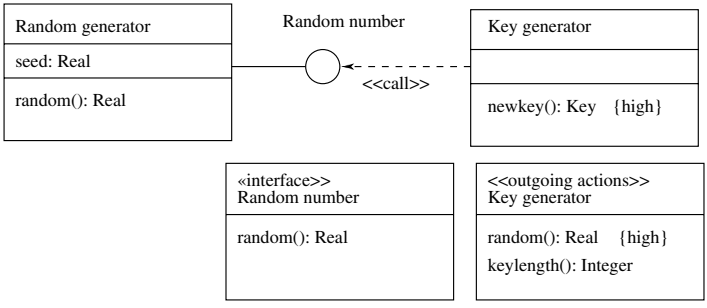


Fig. 2. Key generator

4 Interaction

We would like to model security-aspects in particular of distributed objects systems where messages are sent over networks (e. g. the Internet) where they can be intercepted, modified or deleted. Here the security-critical part of the interaction between objects is the exchange of encryption keys etc. by means of cryptographic protocols. Since it is not always possible to use protocols off-the-sheld, but they have to be adjusted to the specific application domain (cf. the example below), and because vulnerabilities often arise at the boundary between a protocol and the rest of the system [Aba00], we would like to specify cryptographic protocols within the general framework of a security-enhanced UML, which we do using using sequence diagrams.

We concentrate on one kind of interaction diagram, the sequence diagrams (collaboration diagrams are very similar). We first give an abstract syntax of aspects of sequence diagrams that allow specification of security-critical interaction between objects (specifically employing cryptography). We proceed to give a formal semantics tailored to our purposes. Again, a more general account can be found elsewhere [IT95]. We concentrate on concurrent objects that each have their own lifelines and exchange messages asynchronously (by sending signals). As pointed out in [BGH$^+$98], sequential systems are a special case of concurrent ones, and synchronous communication (i. e. operation calls) can be modeled using handshake.

A *sequence diagram* $S = (\mathsf{Obj}, \mathsf{MsgSpecs})$ is given by a list Obj of pairs $(\mathsf{obj\_name}, \mathsf{cls\_name})$ (where $\mathsf{obj\_name}$ may be an empty string) and a list of message specifications $\mathsf{MsgSpecs}$. A *message specification* $m = (\mathsf{sender}, \mathsf{receiver}, \mathsf{guard}, \mathsf{signal}(\mathsf{args}))$ consists of the names of the sender and the $\mathsf{receiver}$, a guard, the name of the signal sent and a list of arguments args. We assume that the messages can be ordered linearly, following the view that two events never occur at exactly the same time [RJB99, p.440]; the more general case follows by using a non-deterministic scheduler. As above, a *guard* is a function $g : \mathsf{Assigments} \to \mathsf{Bool}$ evaluating each assignment to a boolean value (where an assignment is any partial function $f : \mathbf{Var} \rightharpoonup \mathbf{Exp}$ similar to above). Here we assume the set $\mathbf{Var}$ to consist of the variables $\mathsf{msg}_{(m,n)}$ representing the n^{th} argument of the signal with name m received most recently.

We define the data type **Exp** of cryptographic messages that can be exchanged during the interaction. We assume a set $\mathcal{D}$ of basic data values. The set **Exp** contains the expressions defined inductively by the grammar

$E ::=$	expression
d	data value ($d \in \mathcal{D}$)
K	key ($K \in$ **Keys**)
x	variable ($x \in$ **Var**)
$E_1 :: E_2$	concatenation
$\{E\}_e$	encryption ($e \in$ **Keys** $\cup$ **Var**)
$\mathcal{D}ec_e(E)$	decryption ($e \in$ **Keys** $\cup$ **Var**)

Here we consider asymmetric encryption, so the set **Keys** is the disjoint union of the sets of private and public keys, where the private keys corresponding to a public key K is denoted K^{-1}. We assume $\mathcal{D}ec_{K^{-1}}(\{E\}_K) = E$, and for the following example (for RSA signing) also $\{\mathcal{D}ec_{K^{-1}}(E)\}_K = E$ (for all $E \in$ **Exp**, $K, K^{-1} \in$ **Keys**), and that no other equations except those implied by these hold. We write **Msg** for the set of *messages* of the form $\mathsf{sig}(E_1, \ldots, E_n)$ where sig is a signal with n arguments of type **Exp**.

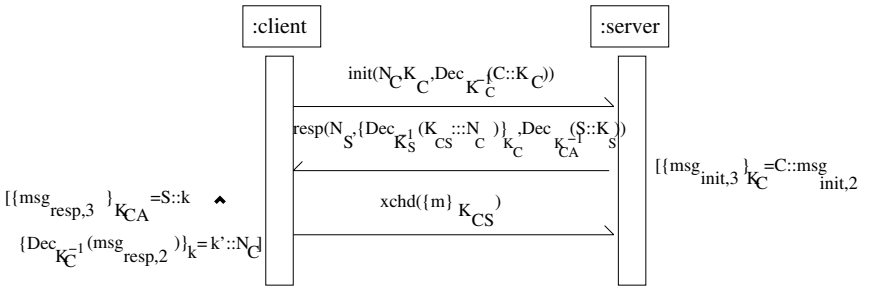


Fig. 3. Variant of TLS

Example: Proposed variant of TLS. The protocol in Figure 3 has been proposed in [APS99] as a variant of the handshake protocol of TLS (the successor of the Internet protocol SSL) to satisfy certain performance constraints (for more details cf. [Jür01c]).

4.1 Interpreting Sequence Diagrams

To specify security properties we give a formal interpretation of sequence diagrams in the specification framework Focus [BS00] which was suggested in [BGH+98] for a formal semantics of UML and was used e. g. in [Jür01c,AJ00] to reason about security. In Focus, one can model concurrently executing systems interacting by transmitting sequences of data values over unidirectional

FIFO communication channels. Communication is asynchronous in the sense that transmission of a value cannot be prevented by the receiver. Focus uses streams and stream-processing functions defined in the following.

For a set C we write $\mathbf{Stream}_C \stackrel{\text{def}}{=} (\mathbf{Msg}^\omega)^C$ for the set of C-indexed tuples of (finite or infinite) sequences of messages, the (untimed) *streams*. A function $f : \mathbf{Stream}_I \to \mathcal{P}(\mathbf{Stream}_O)$ from streams to sets of streams represents a reactive system that receives an input stream $\boldsymbol{s}$ on the input channels in I and nondeterministically sends out an output stream (from the set of possible output streams $f(\boldsymbol{s})$) on the output channels in O.

The composition $f_1 \otimes f_2 : \mathbf{Stream}_I \to \mathcal{P}(\mathbf{Stream}_O)$ of two functions $f_i : \mathbf{Stream}_{I_i} \to \mathcal{P}(\mathbf{Stream}_{O_i})$ $(i = 1, 2)$ (with $O_1 \cap O_2 = \emptyset$, $I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$ and $O = (O_1 \cup O_2) \setminus (I_1 \cup I_2)$), linking input and output channels, is defined by $f_1 \otimes f_2(\boldsymbol{s}) \stackrel{\text{def}}{=} \{\boldsymbol{t}\lfloor_O : \boldsymbol{t} \in \mathbf{Stream}_{I \cup O} \wedge \boldsymbol{t}\lfloor_I = \boldsymbol{s}\lfloor_I \wedge \forall i . \boldsymbol{t}\lfloor_{O_i} \in f_i(\boldsymbol{s}\lfloor_{I_i})\}$.

Interpretation. Again we can only sketch the interpretation, details are in [Jür01b]. Suppose we have a sequence diagram $S = (\mathsf{Objects}, \mathsf{Messages})$. For every $O \in \mathsf{Object}$ we will define a function $[\![O]\!] : \mathbf{Stream}_{\{\mathsf{in}_O\}} \to \mathcal{P}(\mathbf{Stream}_{\{\mathsf{out}_O\}})$: Let $\mathsf{MsgSpecs}_O$ be the sublist of $\mathsf{MsgSpecs}$ of message specifications with O as sender. Any $\boldsymbol{s} \in \mathbf{Stream}_{\{\mathsf{in}_O\}}$ defines an assignment $\mathsf{ass}(\boldsymbol{s})$ by sending the variable $\mathsf{msg}_{(m,n)}$ to the n^{th} argument of the signal named m received most recently. Let $[\![O]\!](\boldsymbol{s})$ be the singleton set consisting of the sequence of those messages $\mathsf{signal}(\mathsf{arguments})$ from $\mathsf{MsgSpecs}_O$ for which $g(\mathsf{ass}(\boldsymbol{s})) = \mathsf{true}$ for the corresponding guard g.

4.2 Secrecy

We say that a stream-processing function $f : \mathbf{Stream}_I \to \mathcal{P}(\mathbf{Stream}_O)$ *may eventually output* an expression $E \in \mathbf{Exp}$ if there exist streams $\boldsymbol{s} \in \mathbf{Stream}_I$ and $\boldsymbol{t} \in f(\boldsymbol{s})$, a channel $c \in O$ and an index $j \in \mathbb{N}$ such that E is an argument of the message $(\boldsymbol{t}(c))_j$.

The following definition uses the notion of an *adversary* from [Jür01c], which is a specific kind of stream-processing function that captures the capabilities of an adversary intercepting the communication link between the distributed objects (the exact definition of "adversary" and "without access" have to be left out here for space limitations but can be found in [Jür01c]).

Definition 3 *We say that a system modeled by a sequence diagram S* leaks *a secret $d \in \mathcal{D} \cup \mathbf{Keys}$ if there is an adversary A without access to d such that $[\![S]\!] \otimes [\![A]\!]$ may eventually output d. Otherwise we say that S* preserves the secrecy of *d.*

S preserves the secrecy of d if no adversary can find out d in interaction with the system modeled by S, following the approach of Dolev and Yao (1983), cf. [Aba00,Jür01c].

Example (continued). It has been shown in [Jür01c] that the proposed variant of the TLS handshake protocol given above does not preserve the secrecy of d.

5 Physical View

We give the abstract syntax of (aspects of) deployment models, extended with the tag {high} used to ensure that security requirements on communication links between different components are met by the physical layer.

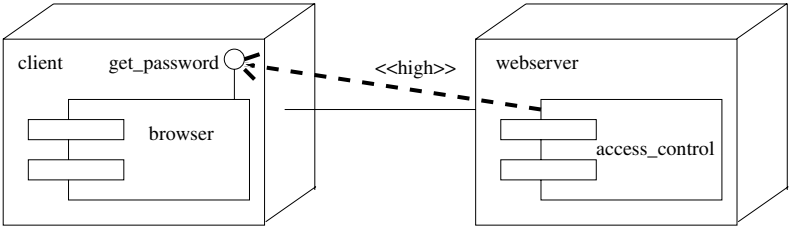
A *component* $C = (\mathsf{comp\_name}, \mathsf{Interfaces})$ is specified by its name comp_name and a (possibly empty) set of Interfaces.

A *node* $N = (\mathsf{node\_name}, \mathsf{Components})$ is given by a name node_name and a set of contained Components.

A *deployment diagram* $D = (\mathsf{Nodes}, \mathsf{Links}, \mathsf{Dependencies})$ is given by a set of Nodes, a set of Links and a set of Dependencies. A *link* $l = (\mathsf{nds}, \mathsf{target})$ is given by a two-element set nds of nodes being linked and a set of tags (that may or may not contain the tag {high} indicating the corresponding security level of the link). Here a *dependency* is a tuple $(\mathsf{client}, \mathsf{supplier}, \mathsf{interface}, \mathsf{tags})$ consisting of component names client and supplier, an interface name interface and a set of tags giving the security level of the dependency. We assume that for every dependency $D = (C, S, I, t)$ there is exactly one link $L_D = (N, t')$ such that $N = \{C, S\}$ for the set of linked nodes.

Definition 4 *A deployment diagram provides* communication security *if for each dependency D that is tagged* {high}*, the corresponding link* L_D *is also tagged* {high}.

Example



This model does not provide communication security, because the communication link between web-server and client does not provide the needed security level.

6 Conclusion and Future Work

The aim of this work is to use UML to encapsulate knowledge on prudent security engineering and to make it available to developers not specialised in security by highlighting aspects of a system design that could give rise to vulnerabilities.

Concentrating on the kinds of diagrams most important for our purpose we showed how UML, by using its extension mechanisms, can be used to express standard concepts from formal methods regarding multi-level secure systems and security protocols. These definitions evaluate diagrams of various kinds and indicate possible weaknesses.

In absence of a general formal semantics we use a specifically defined one which covers the aspects needed and to the required degree of detail. It is not a replacement for a general formal semantics but rather exemplifies the need for one.

In further work [Jür01a], we have used UMLsec to reason about audit-security in a smart-card based payment scheme.

In future work, we will consider the remaining kinds of diagrams and move closer to the unabridged standard of UML by bringing in more detail. In particular, we will explore the use of profiles or prefaces [CKM+99]. Also, we will try to link the various views on a system given by different diagrams to gain additional information on the system that may be security-relevant. To increase usability, we will consider how to automatically derive security annotations from usual modelling information.

To enable tool support, we aim to find efficiently checkable conditions equivalent to the ones given here and to give proof-techniques.

Further case-studies are planned, e. g. regarding development of multilateral security platforms such as [PSW+98].

Acknowledgements. This idea for this work arose when doing security consulting for a project lead by S. Nagaraswany during a research visit with M. Abadi at Bell Labs (Lucent Tech.), Palo Alto, whose hospitality is gratefully acknowledged. This work was presented at the summer school "Foundations of Security Analysis and Design 2000" (Bertinoro), the Computing Laboratory at the University of Oxford, the Department of Computer Science at the TU München, and the Department of Computer Science at the TU Dresden. Comments from S. Abramsky, C. Bolton, M. Broy, J. Davis, G. Lowe, H. Muccini, G. Wimmel, the anonymous referees, and especially A. Pfitzmann, B. Rumpe and P. Stevens are gratefully acknowledged.

References

[Aba00] M. Abadi. Security protocols and their properties. In F.L. Bauer and R. Steinbrueggen, editors, *Foundations of Secure Computation*, pages 39–60. IOS Press, 2000. 20th Int. Summer School, Marktoberdorf, Germany.

[AJ00] M. Abadi and Jan Jürjens. Formal eavesdropping and its computational interpretation, 2000. submitted.

[And94] R. Anderson. Why cryptosystems fail. *Communications of the ACM*, 37(11):32–40, November 1994.

[APS99] V. Apostolopoulos, V. Peris, and D. Saha. Transport layer security: How much does it really cost ? In *Conference on Computer Communications (IEEE Infocom)*, New York, March 1999.

[AR00] E. Astesiano and G. Reggio. Formalism and method, 2000. to appear in Theoretical Computer Science.

[BAN89] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989.

[BD00] C. Bolton and J. Davies. Using relational and behavioural semantics in the verification of object models. In C. Talcott and S. Smith, editors, *Proceedings of FMOODS*. Kluwer, 2000.

[BGH+98] R. Breu, R. Grosu, F. Huber, B. Rumpe, and W. Schwerin. Systems, views and models of UML. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, 1998.

[BS00] M. Broy and K. Stølen. *Specification and Development of Interactive Systems*. Springer, 2000. (to be published).

[CKM+99] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. Defining UML family members using prefaces. In Ch. Mingins and B. Meyer, editors, *TOOLS'99 Pacific*. IEEE Computer Society, 1999.

[DS00] P. Devanbu and S. Stubblebine. Software engineering for security: a roadmap. In *The Future of Software Engineering*, 2000. Special Volume (ICSE 2000).

[EFLR99] A. Evans, R. France, K. Lano, and B. Rumpe. The UML as a formal modeling notation. In J. Bezivin and P.-A. Muller, editors, *The Unified Modeling Language - Workshop UML'98: Beyond the Notation*, LNCS. Springer, 1999.

[For99] UML Revision Task Force. OMG UML Specification 1.3. Available at http : //www.omg.org/uml, 1999.

[GM82] J. Goguen and J. Meseguer. Security policies and security models. In *Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, 1982.

[GPP98] M. Gogolla and F. Parisi-Presicce. State diagrams in UML: A formal semantics using graph transformations. In M. Broy, D. Coleman, T. Maibaum, and B. Rumpe, editors, *PSMT'98*. TU München, TUM-I9803, 1998.

[Huß99] H. Hußmann. Formale Beschreibungstechniken und praktische Softwaretechnik – eine unglückliche Verbindung ? In K. Spies and B. Schätz, editors, *Formale Beschreibungstechniken '99*, pages 1–6. Herbert Utz Verlag, 1999.

[IT95] ITU-T. Z.120 B – Message Sequence Chart Algebraic Semantics. ITU-T, Geneva, 1995.

[Jür00] Jan Jürjens. Secure information flow for concurrent processes. In C. Palamidessi, editor, *CONCUR 2000 (11th International Conference on Concurrency Theory)*, volume 1877 of *LNCS*, pages 395–409, Pennsylvania, 2000. Springer.

[Jür01a] Jan Jürjens. Object-oriented modelling of audit security – a smart-card case study. 2001. submitted.

[Jür01b] Jan Jürjens. *Principles of Secure Systems Design*. PhD thesis, Oxford University Computing Laboratory, 2001. in preparation.

[Jür01c] Jan Jürjens. Secrecy-preserving refinement. In J. Fiadeiro and P. Zave, editors, *Formal Methods Europe*, LNCS. Springer, 2001. to be published.

[Low96] G. Lowe. Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR. In Margaria and Steffen, editors, *TACAS*, volume 1055 of *LNCS*, pages 147–166. Springer, 1996.

[Öve00] G. Övergaard. Formal specification of object-oriented meta-modelling. In *FASE2000*, volume 1783 of *LNCS*. Springer, 2000.

[PSW+98] A. Pfitzmann, A. Schill, A. Westfeld, G. Wicke, G. Wolf, and J. Zöllner. A Java-based distributed platform for multilateral security. In *IFIP/GI Working Conference "Trends in Electronic Commerce"*, volume 1402 of *LNCS*, pages 52–64. Springer, 1998.

[RACH00] G. Reggio, E. Astesiano, C. Choppy, and H. Hußmann. Analysing UML active classes and associated state machines – A lightweight formal approach. In *FASE2000*, volume 1783 of *LNCS*. Springer, 2000.

[RCA00] G. Reggio, M. Cerioli, and E. Astesiano. An algebraic semantics of UML supporting its multiview approach. In D. Heylen, A. Nijholt, and G. Scollo, editors, *AMiLP 2000*, 2000.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[RWW94] A. Roscoe, J. Woodcock, and L. Wulf. Non-interference through determinism. In *ESORICS 94*, volume 875 of *LNCS*. Springer, 1994.

[SP00] P. Stevens and R. Pooley. *Using UML*. Addison-Wesley, 2000.

Grammar Testing

Ralf Lämmel

CWI, Kruislaan 413, NL-1098 SJ Amsterdam
Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam
Email: ralf@cwi.nl
WWW: http://www.cwi.nl/~ralf/

Abstract. Grammar testing is discussed in the context of grammar engineering (i.e., software engineering for grammars). We propose a generalisation of the known rule coverage for grammars, that is, context-dependent branch coverage. We investigate grammar testing, especially coverage analysis, test set generation, and integration of testing and grammar transformations. Grammar recovery is chosen as a subfield of grammar engineering to illustrate the developed concepts. Grammar recovery is concerned with the derivation of a language's grammar from some available resource such as a semi-formal language reference.

1 Introduction

Grammar engineering. Grammars are omnipresent in software development. They are used for the definition of syntax, exchange formats and others. Several important kinds of software rely on grammars, e.g., compilers, debuggers, profilers, slicing tools, pretty printers, (re-) documentation tools, language reference manuals, browsers and IDEs, software analysis tools, code preprocessing tools, and software modification tools. The common practice of grammar development does not quite reflect this importance of grammars. In practice, the most important property of a grammar is to be implementable with a certain tool (e.g., YACC). This focus leads to what we call "grammar hacking", where issues like testing and disciplined adaptation of grammars play a minor role. Companies, which crucially rely on grammars, suffer from this lack of engineering discipline. Grammar hacking now sees a renaissance in the XML-age. We take a different point of view: Grammars are regarded as proper software engineering artifacts (cf. [15,3,12,5] for a similar attitude). We envision *grammar engineering*, where the development, the maintenance, the recovery, and the implementation of grammars are based on well-founded concepts. The *technique* of generalised LR-parsing [14] (as opposed to LALR(1)), for example, covers the full class of context-free grammars. Thereby, it enables a declarative style of grammar specification, and it removes the gap between specification and (YACC-) implementation. Thus, generalised LR-parsing facilitates the *method* of modular syntax definition because the absence of grammar class restrictions allows us to take unions of grammar modules.

Grammar testing. There are several concepts which contribute to grammar engineering. Not all of them have been sufficiently covered in the literature or

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 201–216, 2001.

adopted as common practice. Fields of grammar engineering which are meanwhile reasonably understood are, for example, grammar implementation and grammar reengineering [15]. Fields which lack fundamental underpinnings are grammar adaptation, grammar maintenance, grammar recovery, and grammar testing. The present paper is concerned with grammar testing. From a software engineering perspective, grammars are specifications (describing languages), or programs (serving as compiler compiler input and others). Testing is one standard way to validate specifications or programs (formal analysis is another). The approach to grammar testing described in the sequel covers various technical and pragmatic aspects such as coverage notions, test set generation, correctness and completeness claims for grammars or parsers, and integration of testing and grammar transformations.

Structure of the paper. In Section 2, the contributions of the paper are motivated by a subfield of grammar engineering, namely grammar recovery. In the two subsequent technical sections, grammar recovery serves for illustrative purposes. In Section 3, the notion of context-dependent branch-coverage is developed. It improves on the previously known notion of rule coverage. In Section 4, derived concepts for grammar testing are discussed, e.g., automated coverage analysis and test set generation. In Section 5, the paper is concluded.

Acknowledgement. This work was supported, in part, by *NWO*, in the project "*Generation of Program Transformation Systems*". I am grateful for the comments by the FASE'01 anonymous referees. I am also grateful to Jeremy Gibbons, Jörg Harm, Jan Heering, Paul Klint, Chris Verhoef and Dave Wile for encouraging discussions, and for helpful suggestions. Some results from the present paper have been presented at the 54th IFIP WG2.1 ("Algorithmic Languages and Calculi") meeting in Blackheath, London, April, 3th–7th, 2000, and at the 2nd Workshop Software-Reengineering, Bad Honnef, May, 11th–12th, 2000.

2 Motivation: Grammar Recovery

The following discussion of grammar recovery motivates grammar testing and other grammar engineering concepts. Grammar recovery comprises the concepts involved in the derivation of a language's grammar from some available resource like a language reference, or compiler source code. In [12], a more general account on grammar recovery is given.

2.1 One Scenario

There are several sensible scenarios for grammar recovery. In the present paper, we restrict ourselves to the scenario characterised as follows.

Lack of an approved grammar for a language L. If we want to renovate code in some language L, e.g., in a dialect of an ancient language like COBOL, a suitable grammar is a pre-condition for certain forms of tool support, e.g., grammar-based software modification tools. The compiler vendor might not provide us with the grammar for the language, or the sources for the compiler. Also, the sources

might not be immediately useful for an extraction of the grammar, e.g., in the case of manually implemented frontends. It is also conceivable that the compiler source code and the vendor are not accessible altogether.

Approximative grammar γ_0 for the intended language L. We assume that there is an approximative grammar γ_0 for L. The grammar could be obtained from a semi-formal language reference, or some grammar-based tool the sources of which are available. By *approximative* we mean that we accept that γ_0 is probably incomplete and incorrect w.r.t. L to a certain extent. This is the challenging aspect of the scenario. Thereby, grammar recovery goes beyond grammar reengineering in the narrow sense, where one is merely concerned with the extraction of the base-line grammar from the sources of a reference implementation, and its refactoring [15]. Let us define the terms incorrectness and incompleteness.

Definition 1. *A grammar G* is incorrect w.r.t. an (intended) language L, *if $L \not\supseteq \mathcal{L}(G)$. Here, $\mathcal{L}(G)$ denotes the language generated by G, i.e., all terminal strings derivable from the start symbol of G. The grammar G* is incomplete w.r.t. L *if $L \not\subseteq \mathcal{L}(G)$.*

The definition is inspired by a point of view as favoured in the diagnosis of logic programs (cf. [6]). The problem with the definition is that the intended language L is not directly accessible. Also, we need to know how to locate incorrect or incomplete phrases in the grammar, and how to correct and complete them. Finally, incorrectness and incompleteness are intertwined. An incorrect phrase often causes some incompleteness. Correcting the phrase also contributes to the completion of the grammar. We should point out that our model of correct and complete grammars is a bit naive (for brevity). In practice, one has usually to consider correctness and completeness of *parsers*. Thereby, further requirements might be implied, e.g., suitable and correct parse trees, grammar class restrictions, disambiguation, and others.

Representative trusted test set C written in L. By *trusted* we mean that the code base C is known to comply with L because it is accepted by a reference implementation I such as a parser or a compiler for L. Note that we do not necessarily need I physically. It might also be sufficient if C was approved by someone else who applied I, or by the corresponding standardisation board, or by the compiler vendor. Besides legal issues, it is not clear if re-compilation is a valid option in order to obtain a useful grammar from a compiler or another language implementation, if it is only available in binary form. By *representative* we mean that C should be large enough to experience all possible constructs of the intended language L in some sense.

Derivation of a relatively complete and correct grammar γ_n for L. The overall idea is to parse more and more programs from C, and to resolve the upcoming incompleteness problems indicated by failure of parsing by specific grammar transformations, e.g., by generalising phrases in the grammar accordingly. Due to the intertwined character of incompleteness and incorrectness, the correctness will also be improved in that process because usually the grammar is modified

and not just extended. This stepwise process which results in intermediate grammars $\gamma_1, \gamma_2, \ldots$ is stopped with γ_n if C has been parsed. In each step, a grammar transformation t_i is used to address a particular error or omission. Since we consider γ_0 as an approximation of L, the structure of γ_0 should be preserved in γ_n as much as possible.

2.2 Case Study

We use VS COBOL II as case study. Its grammar was recovered from IBM's reference [10]. As for the test set involved in the project, 2 millions lines of code from several software projects were used. The raw grammar extracted from IBM's reference was corrected and completed by about 300 small transformation steps. The recovered grammar has been published in December 1999 [11]. It is used by COBOL practioners, tool developers, compiler writers and others since then. The grammar is the first publically available quality COBOL grammar. It was obtained in just a few weeks based on a reproducable process for grammar recovery described in [12]. This figure is in strong contrast to other known figures for (unpublished) quality COBOL grammars. It takes some engineering knowledge or a lot of money to come up with good grammars.

From diagrams to the raw grammar. The IBM reference contains large portions of the VS COBOL II syntax in a graphical notation of so-called syntax diagrams. Some diagrams are not even syntactically correct in the sense of the underlying diagram notation. Other diagrams are semantically incorrect in the sense that they do not generate the intended language. Many diagrams are not correct and complete in the sense that the reference uses informal notes to relax or to restrict the diagrams. Using a dedicated parser for syntax diagrams, the raw VS COBOL II grammar was extracted from the reference. This raw grammar provides the initial and approximative grammar γ_0 according to the scenario sketched above. A fragment of the raw VS COBOL II grammar is shown in Fig. 1 using extended BNF notation.

```
Data-description-entry =
  Level-number (Data-name | "FILLER")? Redefines-clause?
  Blank-when-zero-clause? External-clause? Global-clause?
  Justified-clause? Occurs-clause? Picture-clause? Sign-clause?
  Synchronized-clause? Usage-clause? Value-clause?
Redefines-clause =
  Level-number (Data-name | "FILLER")? "REDEFINES" Data-name
```

Fig. 1. Fragment of raw grammar extracted from [10]

Incorrectness. The fragment in Fig. 1 provides a good example for an incorrectness of the raw extracted grammar. The format of `Redefines-clause` does actually cover more than just the structure of the clause itself. We might assume that the correction is performed by a grammar transformation which simply

deletes the obsolete part of the incorrect definition from the raw grammar. The proper definition `Redefines-clause` is the following:

```
Redefines-clause = "REDEFINES" Data-name
```

Error detection by parsing. In general, an incorrectness might go undetected if only parsing is used for diagnosis because incorrectness means that the language generated by a grammar contains words which are not contained in the intended language. The above incorrectness (and many others) will be realised when approved code is parsed, if the code experiences `Redefines-clause`. This is implied by the intertwined character of incorrectness and incompleteness. The incorrect definition of `Redefines-clause` is *in conflict* with the correct form.

Incompleteness. The most basic form of incompleteness is that some nonterminals are not defined altogether. This form of incompleteness can be recognized statically. Otherwise, incompleteness means that certain phrases in the grammar are too restricted, i.e., they do not cover the corresponding construct in full generality. This form of incompleteness can be uncovered by parsing if the test set experiences the corresponding construct as assumed for a representative test set. The IBM reference, for example, contains an informal note that the order of the clauses in a data description entry is immaterial. In fact, actual COBOL code experiences different orders. The sequence of clauses in Fig. 1 should be turned into a permutation phrase.

2.3 Challenges

There are certain questions the answers to which affect the process of grammar recovery in an essential manner. How do we assess the quality of the code base C? How do we precisely know that C is representative? What does it mean for γ_n to be complete and correct? A completeness relative to C can be claimed if $C \subseteq \mathcal{L}(\gamma_n)$, that is, if C can be parsed. According to Definition 1, correctness in a narrow sense means that $L \supseteq \mathcal{L}(\gamma_n)$. However, L is not directly accessible. So what is a realistic requirement for the relative correctness of γ_n? Besides completeness and correctness of γ_n, what is the relationship between γ_0 and γ_n? The latter question is concerned with the stepwise process of correction and completion. In general, we might ask what kind of properties can be required for the transformations t_i for the steps. What does it mean for t_i to correct or to complete resp. the grammar?

In order to answer these questions thoroughly, concepts for testing grammars are worked out in the following two sections. These concepts are not only useful for grammar recovery but also for other grammar-dependent problems, e.g., parser testing, grammar maintenance, and automated software modification.

3 Context-Dependent Branch Coverage

We discuss the notion of context-dependent branch coverage obtained as an essential generalisation of rule coverage [13]. Firstly, rule coverage will be revisited. Secondly, a context-dependent generalisation is developed. Finally, the use of the coverage notion is illustrated in grammar recovery.

3.1 Rule Coverage

Rule coverage simply means that a test set explores all rules of a grammar. It is clear that for each reduced context-free grammar a finite test set achieving coverage of all rules exists. Note that we assume non-ambiguous grammars in the following definition of rule coverage, and also in most subsequent definitions.

Definition 2. *Let $G = \langle N, T, s, P\rangle$ be a context-free grammar. $w \in \mathcal{L}(G)$ is said to* cover *$p = n \to u \in P$ if there is a derivation $s \Rightarrow_G^* x\ n\ y \overset{p}{\Rightarrow}_G x\ u\ y \Rightarrow_G^* w$. $W \subseteq \mathcal{L}(G)$ is said to* achieve rule coverage (RC) for *G, if for each $p \in P$ there is a $w \in W$ which covers p.*

Application to parser testing. Let us provide a scenario where rule coverage can be used in grammar implementation. Consider a parser P which is assumed to implement a grammar G. Let us assume that the implementation even follows the structure of G, e.g., by using recursive-descent parsing. If G is a non-ambiguous grammar, and W is a test set achieving rule coverage for G, then parsing W with P implies that the parser has to experience all rules of G. Major implementational defects, for example, in the sense of incompleteness of P will be detected in this way. These defects will be reported by the failure of P for certain elements from W. The incorrectness of P could be detected using negative test cases covering mutations of G using ideas from mutation testing [9].

G_1	G_2	G_3	G_4
$[r_1]\ s \to A\ B$	$[r_1]\ s \to A\ B$	$[r_1]\ s \to A\ B$	$[r_1]\ s \to A\ B$
$[r_2]\ A \to C\ a$	$[r_2]\ A \to a$	$[r_2]\ A \to C'\ a$	$[r_2]\ A \to C\ a$
$[r_3]\ B \to b\ C$	$[r_3]\ B \to b\ C$	$[r_3]\ B \to b\ C'$	$[r_3]\ B \to b\ C'$
$[r_4]\ C \to \epsilon$	$[r_4]\ C \to \epsilon$	$[r_4]\ C \to \epsilon$	$[r_4]\ C \to \epsilon$
$[r_5]\ C \to c\ C$	$[r_5]\ C \to c\ C$	$[r_5]\ C \to c\ C$	$[r_5]\ C \to c\ C$
		$[r_6]\ C' \to C$	$[r_6]\ C' \to C$

Fig. 2. Sample grammars ($\mathcal{L}(G_1) = \mathcal{L}(G_3) = \mathcal{L}(G_4) \supseteq \mathcal{L}(G_2)$)

Example 1. The test set $\{a\,b\,c\}$ covers all rules of G_1 from Fig. 2 whereas $\{a\,b\}$ does not because rule $[r_5]$ is not covered.

Limitations. It is clear that for any coverage notion, in principle, one can construct grammars G and G' with $\mathcal{L}(G) \not= \mathcal{L}(G')$ so that the difference is not uncovered solely based on test sets achieving coverage. This is implied by decidability results for context-free languages, and also by the fact that test sets have to be finite. In a pragmatic sense, we are looking for a *powerful* coverage notion which is useful in practice to uncover major differences between grammars. Rule coverage is by far not sufficient for that purpose.

Example 2. G_1 and G_2 from Fig. 2 do not generate the same language because G_2 lacks the occurrence of C in $[r_2]$. Note that $\mathcal{L}(G_1) \supset \mathcal{L}(G_2)$ because ϵ is derivable from C. The set $\{a\,b\,c\}$ covers all rules of G_1 and G_2. Thus, rule coverage does not uncover the incompleteness of G_2 w.r.t. G_1.

Let us rephrase the above example in the context of parser testing. Suppose, G_1 serves as specification, but an actual parser accidentally implements G_2. A test set achieving rule coverage for the specification G_1 does not necessarily uncover the incompleteness of the parser implementing G_2. Any testing method as opposed to formal verification is limited in the sense that errors can only be found to a certain extent. The above example illustrates that rule coverage explores a grammar's structure in a rather weak sense.

3.2 Context-Dependent Rule Coverage

We propose a generalisation of rule coverage, where the context in which a rule is covered is taken into account. This idea is easy to formalise.

Definition 3. *Let $G = \langle N, T, s, P\rangle$ be a context-free grammar. If $m \to u\ n\ v \in P$, where $m, n \in N$, $u, v \in (N \cup T)^\star$, then $m \to u\ \boxed{n}\ v$ is called* direct occurrence of n in G. $\mathsf{Occs}(G, n)$ *denotes* the set of all direct occurrences of n in G.

Example 3. All direct occurrences of G_1 from Fig. 2 are listed:

$$\begin{aligned}
\mathsf{Occs}(G_1, s) &= \emptyset \\
\mathsf{Occs}(G_1, A) &= \{[r_1]\ s \to \boxed{A}\ B\} \\
\mathsf{Occs}(G_1, B) &= \{[r_1]\ s \to A\ \boxed{B}\} \\
\mathsf{Occs}(G_1, C) &= \{[r_2]\ A \to \boxed{C}\ a, [r_3]\ B \to b\ \boxed{C}, [r_5]\ C \to c\ \boxed{C}\}
\end{aligned}$$

Definition 4. *Let $G = \langle N, T, s, P\rangle$ be a context-free grammar. $w \in \mathcal{L}(G)$ is said to* cover the rule $p = n \to z \in P$ for the occurrence $m \to u\ \boxed{n}\ v$ *if there is a derivation $s \Rightarrow_G^* x\ m\ y \overset{q}{\Rightarrow}_G x\ u\ n\ v\ y \overset{p}{\Rightarrow}_G x\ u\ z\ v\ y \Rightarrow_G^* w$ with $q = m \to u\ n\ v \in P$. $W \subseteq \mathcal{L}(G)$ is said to* cover $p = n \to z \in P$ for all occurrences*, if there is a $w \in W$ for all occurrences $o \in \mathsf{Occs}(G, n)$ such that w covers p for o. W is said to* achieve context-dependent rule coverage (CDRC) for G, *if W covers all $p \in P$ for all occurrences.*

Example 4. CDRC separates the grammars G_1 and G_2 from Fig. 2. Every test set W achieving CDRC for G_1, e.g., $W = \{a\,b, c\,c\,a\,b\,c\}$ must explore both rules for C in all occurrences of C. Thus, the incompleteness of G_2 w.r.t. G_1 is uncovered.

Theorem 1. *For all reduced context-free grammars $G = \langle N, T, s, P\rangle$, if W achieves context-dependent rule coverage of G, then W also achieves rule coverage of G provided the following side condition holds: $\mathsf{Occs}(G, s) \not= \emptyset$, or s is defined by a single rule.*

Proof. In a reduced context-free grammar, each rule $n \to z \in P$ with $n \not= s$ has at least one occurrence. Thereby, the rule is covered per pre-condition (for even all occurrences). If s is defined by a single rule, this rule will be covered since derivation starts from s. Otherwise, per side condition we know that there are occurrences of s. Then, the rules defining s are covered like all other rules.

Sensitivity. There is a problem with the context-dependent coverage notion as it stands now. The given definition is very sensitive to chain rules and fold/unfold manipulations. These concepts are useful to improve the structure of a given grammar. By *sensitivity* we mean that the fact if a test set achieves coverage or not is dependent on the existence of chain rules, and it varies for grammars which are equivalent modulo fold/unfold manipulations. First, we identify some relevant terms. Then, we indicate a solution for the sensitivity problem.

Definition 5. *The nonterminal n is said to be* non-branching *in a context-free grammar G if there is exactly one defining rule for n in G. $\mathcal{NB}(G)$ denotes the set of non-branching nonterminals in G. A rule is said to be an* injection*, if it is of the form $n \to n'$, where both n and n' are nonterminals. A rule is said to be a* chain rule*, if it is an injection, and the nonterminal on the left-hand side is non-branching.*

Example 5. Although $\mathcal{L}(G_1) = \mathcal{L}(G_3) = \mathcal{L}(G_4)$, CDRC of G_3 can already be achieved with $W = \{a\,b\,c\,c\}$ whereas W is not sufficient for G_1 and G_4. The three grammars are structurally equivalent under chain rule elimination. G_1 does not contain chain rules. G_3 and G_4 contain the chain rule $[r_6]\ C' \ \to\ C$. Note that we can also understand the structural differences between the grammars in terms of fold/unfold manipulations. Coverage for G_3 does not imply that the rules defining C are covered for the *two* occurrences of C'. Therefore, coverage is easier to achieve for G_3. The chain rule in G_4 does not affect coverage because C' has only *one* occurrence, and thus the rules for C will be exhausted via the definition of C'.

The sensitivity of context-dependent rule coverage can be decreased considerably, if non-branching nonterminals are handled in a special way. We can consider indirect occurrences as opposed to direct occurrences in Definition 3 where rules of non-branching nonterminals are involved for intermediate derivation steps. We use sequences of direct occurrences in order to represent indirect occurrences.

Definition 6. *Let G be a context-free grammar. The sequence of direct occurrences $\langle n_0 \to u_1\ \boxed{n_1}\ v_1, \ldots, n_{m-1} \to u_m\ \boxed{n_m}\ v_m, \rangle$ for $m > 1$ is called* indirect occurrence of n_m in G reachable via $M \subseteq N$ *if $n_1, \ldots, n_{m-1} \in M$, and $n_1, \ldots, n_m$ are pairwise distinct.*

We can update the notation $\mathsf{Occs}(G, n)$ to refer to both direct and indirect occurrences. Definition 4 is also easy to generalise. We use CDRC^M to denote the refinement of CDRC to take indirect occurrences reachable via M into account. The described sensitivity problem is solved if $\mathcal{NB}(G)$ is chosen for M.

Example 6. Recall the problem from Example 5. The set $\{a\,b\,c\,c\}$ is not sufficient to achieve CDRC* for G_3, although it was sufficient to achieve CDRC.

Note that the simple CDRC is harder to achieve, if non-branching nonterminals are eliminated before coverage is considered. As for chain rules, this normalisation corresponds to chain rule elimination being one of the steps involved in the folklore algorithm for obtaining Chomsky Normal Form. For more general non-branching nonterminals, a simple unfold step is usually sufficient. The reason why we do not attempt such a normalisation is that we want to consider coverage of the original grammar. If some rule is not experienced in a certain context, for example, then, for traceability, the corresponding analysis should refer to the original grammar rather than to a normalised grammar.

Both rule coverage and the context-dependent generalisation of it were stated for pure context-free grammars, that is, basic BNF notation. Often extended BNF (EBNF) notation is used. One way to look at EBNF is that other constructs for branching than just multiple rules for a nonterminal are provided. In this sense, we are looking for slight generalisations of the defined coverage notions, namely branch coverage and the context-dependent generalisation (CDBC) of it. We do not work out these generalisations in the present paper.

3.3 Application to Grammar Recovery

Recall the scenario for grammar recovery from Section 2.2. It is instructive to notice that an uncovered part of an intermediate grammar γ_i, which accepts some code base C' available at this point in the process, provides an indication of either insufficiency of C' *or* incorrectness of γ_i. If full coverage is achieved, we might claim that both the ultimate C has been accumulated, and a correct and complete γ_n has been found. In practice, it is difficult to accumulate a code base which achieves full coverage, at least for a challenging criterion like CDBC. Thus, in a sense the quality of the code base C' and correctness of γ_i are measured in an intertwined manner.

The value of CDBC. The non-trivial coverage notion CDBC as opposed to simple rule coverage adds precision. Consider, for example, the following three rules $A \to B$, $A \to C$, and $A \to D$ defining A. If in some occurrence, A is used where B, C or D is intended instead, a trusted test set cannot cover the corresponding occurrence. Let us consider an example from the VS COBOL II recovery project. In COBOL, data names can be qualified for nested group fields (records). Moreover, a qualification with a file name can be performed. Thus, we have the following syntax:

```
qualified-data-name =
  data-name (("IN"|"OF") data-name)* (("IN"|"OF") file-name)?
```

IBM's reference for VS COBOL II is imprecise about where non-qualified as opposed to qualified data names are to be used. A decent test set will uncover if `data-name` is too specific for some occurrence. CDBC prevents us from generalising the occurrences of `data-name` more than intended.

4 Grammar Testing

Based on the basic notion of context-dependent coverage introduced in the previous section, further concepts for grammar testing can be supplied. In this section, coverage analysis, test set generation, and integration of testing and transformation are discussed. We want to mention that these concepts for grammar testing are certainly also useful for other formalisms than grammars. We can think of, for example, signatures, algebraic data types, or document type definitions in the XML context.

4.1 Coverage Analysis

Given a test set W and a grammar G, a coverage analyser is supposed to return some representation of the coverage of G by W. As for CDBC, we are interested in the branches which are (not) covered for certain occurrences. We will work out the idea of coverage analysis for basic BNF and direct occurrences. As for the representation of coverage, we assume that the coverage of G by W corresponds to a subset of the full coverage set defined as follows.

Definition 7. *Given a context-free grammar $G = \langle N, T, s, P\rangle$, the* full coverage set $\mathcal{FCS}(G)$ for G *is the following:*

$$\mathcal{FCS}(G) = \left\{ \langle p, o\rangle \;\middle|\; \begin{array}{l} p \in P, \\ o \in \mathsf{Occs}(G, n), \textit{where } p \textit{ is of the form } n \to u \end{array} \right\}$$

Derivation of coverage analysers. We use attribute grammars to formalise coverage analysers. Actually, a scheme to derive an attribute grammar $\mathcal{CA}(G)$ implementing the *c*overage *a*nalyser for G is supplied. Essentially, $\mathcal{CA}(G)$ synthesizes a coverage set from a given test set W. Therefore, W is parsed as a list by $\mathcal{CA}(G)$. The context-free grammar underlying $\mathcal{CA}(G)$ is essentially G but with a new start symbol s' to model lists of words in the sense of G:

$$\langle N \cup \{s'\}, T, s', P \cup \{s' \to \epsilon, s' \to s\ s'\}\rangle$$

We want to synthesize an attribute c for all non-terminals to capture the coverage of G by the parsed test set. All nonterminals n (but s') carry an inherited attribute o of type $\mathsf{Occs}(G, n)$ where we consider a special value $\square$ to encode the top level application of rules for s. The following computations are associated with the rules defining the new start symbol s'.

$$\begin{array}{l} s' \to \epsilon \\ \quad s'.c := \emptyset \end{array} \qquad\qquad \begin{array}{l} s'_0 \to s\ s'_1 \\ \quad s.o\ := \square \\ \quad s'_0.c := s.c \cup s'_1.c \end{array}$$

Their interpretation is straightforward. The coverage of the empty list is the "empty coverage", whereas the coverage of a non-empty list is obtained by taking the union of head's and tail's coverage. The computation for the rules in P follow

a common scheme. Given a rule $p \in P$ with $p = l \to x_1 \cdots x_n$ and $l \in N$, $x_i \in N \cup T$ for $i = 1, \ldots, n$, the following computations are associated with p:

$$\begin{aligned} x_{i_1}.o &:= l \to x_1 \cdots x_{i_1-1} \boxed{x_{i_1}} x_{i_1+1} \cdots x_n \\ &\cdots \\ x_{i_m}.o &:= l \to x_1 \cdots x_{i_m-1} \boxed{x_{i_m}} x_{i_m+1} \cdots x_n \\ l.c &:= x_{i_1}.c \cup \cdots \cup x_{i_m}.c \cup \{\langle p, l.o \rangle\} \end{aligned}$$

The $x_{i_1}, \ldots, x_{i_m}$ correspond to the nonterminals on the right-hand side of p. The intention of the computations is simply to propagate the actual occurrence and to combine coverage from all the nonterminals on the right side of a rule while adding the coverage of the particular rule for the inherited occurrence.

Example 7. $\mathcal{CA}(G_1)$ (refer to Fig. 2 for G_1) synthesizes a coverage for $a\,b$ and $c\,a\,b\,c$, where only the element $\langle C \to c\ C, C \to c\ \boxed{C} \rangle$ is missing from the full coverage set $\mathcal{FCS}(G_1)$. Thus, the rule $C \to c\ C$ has not been used for the recursive occurrence of C in $C \to c\ C$.

There are two useful refinements of the above scheme. Firstly, one can count actual applications of rules for the various occurrences. For that purpose, it is sufficient to allow the attributes c to carry multi-sets. Secondly, CDRC^M can be accomplished by propagating indirect occurrences with the attributes o.

4.2 Test Set Generation

A test set generator is a program which computes a test set achieving the desired coverage criterion. Test set generation is useful, for example, in language design, and parser testing. Returning to the application scenario of parser testing in Section 3.1, test set generation automates testing the parser P w.r.t. a reference grammar G.

The generative application of context-free grammars is reasonably understood. Some fundamental algorithms are developed in [13], e.g., shortest derivations to reach a certain nonterminal, and the derivation of tests sets achieving rule coverage. One approach to test set generation is the following. Given a context-free rule, a shortest completion is computed. Thereby, it is possible to compute a small test set of words with short derivations achieving rule coverage. When grammars are applied for the syntax definition of languages, the choice of shortest completions is beneficial for debugging purposes. One can also favour an even smaller test set of words with longer derivations to cover as many rules in one derivation as possible. The context-dependent generalisation of rule coverage does not introduce any complication: Rules occurring in certain contexts are completed instead of just rules.

4.3 Integration with Transformation

Grammars need to be adapted during recovery, maintenance, and elsewhere. Grammar transformations are useful for an operational and formal model of corresponding adaptations. We separate grammar refactorings which do not change

the generated language, and transformations for construction and destruction which go beyond simple semantics-preserving transformations (in terms of the generated language). An application scenario for grammar refactoring is DeYACCification and modularisation which are useful in grammar reengineering in order to go from YACC-like pure BNF notation to a richer notation (cf. [15]) including constructs for extended BNF and modules as, for example, in SDF [8]. The steps to correct or to complete a grammar in grammar recovery can be modelled by constructing or destructing transformations. We want to study the relation between grammar transformations and grammar testing.

An operator suite. Let $\mathcal{G}$ denote the set of all context-free grammars. Then, a grammar transformation is a partial function on $\mathcal{G}$. Partial functions have to be considered because of applicability conditions. The operators are explained by describing their effect on an input grammar $G = \langle N, T, s, P\rangle$. Some transformations are applied in a focus $F \subseteq N$, that is, the transformations are supposed to affect only the definitions of nonterminals F. There are the following operators. introduce n as u adds the rule $n \to u$ to G. The operator is applicable if $n \not\in N$. fold u to n replaces the phrase $u \in (N \cup T)^\star$ in the rules defining F by n, e.g., $n' \to x\,u\,y$ is turned into $n' \to x\,n\,y$. The operator is applicable if n is a non-branching nonterminal defined by the rule $n \to u$. unfold n replaces the right-hand side occurrences of the nonterminal n in the rules defining F by the definition of n, e.g., $n' \to x\,n\,y$ is turned into $n' \to x\,u\,y$ if G contains the rule $n \to u$. The operator is applicable if n is non-branching in P. eliminate n removes all rules defining the nonterminal n from G. The operator is applicable if n is not reachable from s. The operators for introduction, folding, unfolding, and elimination facilitate grammar refactoring without changing the generated language. The remaining two operators are constructive or destructive resp. in the sense that they increase and decrease of the generated language. include u for n adds the rule $n \to u$ to G. The operator is applicable if n is defined in G, that is, there is at least one rule with n on the left-hand side in G. exclude u from n removes the rule $n \to u$ from G. The operator is applicable if there are two or more rules defining n, one of the form $n \to u$.

Coverage-related relations on grammars. The impact of grammar transformations regarding coverage can be conceived in terms of some relations on grammars, e.g., the property if two grammars are covered by the same test sets.

Definition 8. *We use $\mathcal{R}(G)$ to denote the reduced sub-grammar of G which is obtained by removing rules which are not reachable, or which are not terminated. Given two context-free grammars G and G', and a coverage criterion α, we say* α for G implies α for G' *if for $W \subseteq (\mathcal{L}(\mathcal{R}(G)) \cap \mathcal{L}(\mathcal{R}(G')))$ holds that W achieves α-coverage for $\mathcal{R}(G)$ implies W achieves α-coverage for $\mathcal{R}(G')$. G and G' are called* α-equivalent *if α for G implies α for G' and vice versa.*

Properties of transformations. Based on the above grammar relations, transformations can be characterised w.r.t. coverage as follows.

Definition 9. *A partial function $f : \mathcal{G} \nrightarrow \mathcal{G}$ is said to* improve *(v.s.* hamper*) the coverage criterion α if α for G implies α for $f(G)$ (or vice versa for hampering) for all $G \in \mathcal{G}$ where $f(G)$ is defined. If G and $f(G)$ are* α-equivalent*, f is said to* preserve *α.*

The following theorem states properties of the above operators w.r.t. CDRC^M. The overall conclusion is that refactoring of grammars does not cause sensitivity problems for coverage. The constructing and destructing operators hamper or improve resp. coverage, since they are essentially concerned with adding or removing branches.

Theorem 2.

1. fold u to n *and* unfold n *preserve CDRC*.*
2. eliminate n *and* introduce n as u *preserve CDRC and CDRC*.*
3. include u for n *hampers CDRC and CDRC$^{\mathcal{NB}(G)\cup\{n\}}$.*
4. exclude u from n *improves CDRC and CDRC$^{\mathcal{NB}(G)\cup\{n\}}$.*

Proof. (Sketch)

1. Direct occurrences become indirect occurrences and vice versa.
2. $\mathcal{R}(\cdot)$ neutralizes the eliminated / introduced rule.
3. The included rule has to be covered for $\mathsf{Occs}(G, n)$.
4. The excluded rule has not to be covered anymore.

As for the fold operator and the unfold operator, the consideration of CDRC*, is essential according to the discussion in Section 3.2 (cf. Example 5 in particular). Note that the property of the include operator and the exclude operator to hamper or to improve coverage does not hold for CDRC*. We indeed have to consider indirect occurrences reachable via $\mathcal{NB}(G) \cup \{n\}$ because by including or excluding a rule the defined nonterminal n might change its status to be a non-branching or a branching nonterminal, respectively.

4.4 Application to Grammar Recovery

Approval of uncovered branches. At any point in the process, coverage analysis can be used to compute the coverage of γ_i w.r.t. the currently available and parsed code base C'. Still there is the problem that uncovered branches (in some context) are either an indication of the insufficiency of C' or the incorrectness of γ_i. Let us assume that a reference implementation I for the intended language L is available. Test set generation can be used to generate a test set W from γ_i. We might prefer to generate only programs which improve on the coverage of C'. W can be checked if it is contained in the intended language, i.e., if it is parsed by I. Besides the availability of I, the only pre-condition is that the error messages produced by I are sufficient to separate parsing errors and violations of static semantics. If W is accepted by I, then the phrases corresponding to all branches in all contexts are feasible in L. Thereby, we can approve uncovered branches without even relying on a code base experiencing them.

Correctness and completeness. The above approach suggests a correctness claim w.r.t. CDBC. If I accepts W, the grammar γ_n can be said to be correct w.r.t. CDBC. In this claim, C is not involved. By contrast, completeness is relative to C, that is, γ_n is said to be complete w.r.t. C. In a sense, the ultimate code base C is more important for completing γ_0, whereas test set generation is more relevant to claim correctness of γ_n. Of course, even if I is available to gain confidence in the correctness of γ_n by parsing generated test cases, full correctness cannot be claimed. The property $\mathcal{L}(\gamma_n) \subseteq L$ cannot be checked by repeatedly applying I which models membership test for L. For similar reasons, the completeness claim is inherently relative.

Preservation of structure. Assuming that the extracted raw grammar γ_0 is a useful approximation of L, we want to preserve its structure as much as possible. Both destruction and construction, that is, removal and addition of branches, should be defensive. That is enforced if the transformation sequence $t_1, \ldots, t_n$ satisfies the following requirements:

- Incorrect phrases are removed by destructing transformations.
- Missing phrases are added by constructing transformations.
- Branches of the grammar which can be covered are not removed.
- Added branches are ultimately covered.

The relative completeness and correctness claims for γ_n are strengthened in this way because γ_n can be regarded as a refactored variant of γ_0 with some removed and added branches. The removal and the addition of these branches was solely triggered by C. Context-dependency of coverage is relevant here if branches are to be added or removed only for certain occurrences of a nonterminal.

5 Concluding Remarks

Contribution and applications. Context-dependent branch coverage and derived concepts for coverage analysis, test set generation, and the integration of testing and transformation were developed. These testing concepts are meant as a contribution to grammar engineering. The paper illustrated that more involved coverage criteria than the previously known rule coverage are needed. Our examples were concerned with parser testing and grammar recovery. The paper suggested original relative correctness and completenss claims for parsers and grammars. Several further applications are conceivable, e.g., language design, and grammar minimalisation for automated software renovation.

Related work. Certain rather pragmatic forms of coverage analysis have been suggested by others, e.g., in [2], it was suggested to count rule applications for a YACC-grammar. Previous approaches to test set generation (for testing language processors) are based on either rule coverage or randomized test sets (cf. [1]). Burgess [4] compiled a survey on compiler testing. A challenging problem in testing compilers or other language processors is the generation of semantically correct programs. Thereby, generation of test sets and feasibility of coverage is

considerably more complicated. One can think of coverage in two dimensions for the language definitions specified, for exampple, with attribute grammars: a syntactical dimension corresponding to the underlying context-free grammar, and a semantical dimension corresponding to the attributes, their types, and the computations associated with the productions. This issue is examined in some depth in [7] based on a related coverage criterion.

Perspective. We are working on adequate tool support for coverage analysis, coverage visualisation, test set generation, and grammar transformation. This project is challenged by the fact that the tools should scale up for complex grammars and huge test sets. We would also like to apply our concepts to document type definitions in the XML context. At the conceptual level, there are the following directions for future work. Our relative notions of grammar correctness and completeness can definitely be further improved or complemented. One would like to consider metrics, for example, to quantify the structure preservation for the raw grammar γ_0. Our relative correctness claim for γ_n was based on the generation of *positive* test cases from the grammar to see if they are *parsed* by a reference implementation. Dually, we could use *negative* test cases, which had to be *refused* by the reference implementation, for a stronger completeness claim. Negative test cases could be obtained via mutation testing [9].

References

1. D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.
2. A. Boujarwah and K. Saleh. Compiler test suite: evaluation and use in an automated test environment. *Information and Software Technology*, 36(10):607–614, 1994.
3. M. Brand, M. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In S. Tilley and G. Visaggio, editors, *Proceedings of the sixth International Workshop on Program Comprehension*, pages 108–117, 1998.
4. C. J. Burgess. The Automated Generation of Test Cases for Compilers. *Software Testing, Verification and Reliability*, 4(2):81–99, jun 1994.
5. M. de Jonge and J. Visser. Grammars as Contracts. In *Proc. of GCSE 2000*, LNCS, Erfurt, Germany, 2001. Springer-Verlag. to appear.
6. G. Ferrand. The Notions of Symptom and Error in Declarative Diagnosis of Logic Programs. In P. Fritzon, editor, *Automated and Algorithmic Debugging, First International Workshop, AADEBUG'93*, volume 749 of *Lecture Notes in Computer Science*, pages 40–57. Springer-Verlag, 3–5 May 1993.
7. J. Harm and R. Lämmel. Two-dimensional Approximation Coverage. *Informatica*, 24(3), 2000.
8. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
9. W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.
10. IBM Corporation. *VS COBOL II Application Programming Language Reference*, 1993. Release 4, Document number GC26-4047-07.

11. R. Lämmel and C. Verhoef. VS COBOL II Grammar Version 1.0.3. `http://www.cwi.nl/~ralf/grammars`, 1999–2001.
12. R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. Submitted, available at `http://www.cwi.nl/~ralf/`, July 2000.
13. P. Purdom. A sentence generator for testing parsers. *BIT*, 12(3):366–375, 1972.
14. J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.
15. M. Sellink and C. Verhoef. Development, assessment, and reengineering of language descriptions. In J. Ebert and C. Verhoef, editors, *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, pages 151–160. IEEE Computer Society, March 2000.

Debugging via Run-Time Type Checking

Alexey Loginov, Suan Hsi Yong, Susan Horwitz, and Thomas Reps

Computer Sciences Department, University of Wisconsin-Madison
1210 West Dayton Street, Madison, WI 53706 USA
{alexey, suan, horwitz, reps}@cs.wisc.edu
Fax: (608) 262-9777

Abstract. This paper describes the design and implementation of a tool for C programs that provides run-time checks based on type information. The tool instruments a program to monitor the type stored in each memory location. Whenever a value is written into a location, the location's run-time type tag is updated to match the type of the value. Also, the location's static type is compared with the value's type; if there is a mismatch, a warning message is issued. Whenever the value in a location is used, its run-time type tag is checked, and if the type is inappropriate in the context in which the value is being used, an error message is issued. The tool has been used to pinpoint the cause of bugs in several Solaris utilities and Olden benchmarks, usually providing information that is succinct and precise.

1 Introduction

Java programmers have the security of knowing that errors like out-of-bounds array indexes or attempts to dereference a null pointer will be detected and reported at runtime. Java also provides security via its strong type system. For example:

- There are no union types in Java, so it is not possible for a program to write into a field of one type and then access that value via a field of a different type.
- Only very restricted kinds of casting are allowed; for example, it is not possible to treat a pointer as if it were an integer or vice versa.
- When an object is down-cast to a subtype, a run-time check is performed to ensure that the actual type of the object is consistent with the cast.

C and C++ programmers are not so lucky. These languages are more liberal than Java in what they allow programmers to express; the static type system is weaker; and the run-time system provides little in the way of protection from errors caused by misuse of casts, bad pointer dereferences, or array out-of-bounds errors. Programmers can use Purify[1], Safe-C[2], and shadow processing[3] to help detect bad memory accesses, but those tools provide no help with the many additional kinds of errors that can be introduced into C and C++ programs due to their weak type systems.

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 217–232, 2001.

This paper describes the design and implementation of a tool for C programs that performs run-time checks based on type information. The tool instruments a program to monitor the type stored in each memory location (which may differ from the static type of that location due to the use of unions, pointers, and casting). Whenever a value is written into a location, the location's run-time type tag is updated to match the type of the value. Also, the location's static type is compared with the value's type; if there is a mismatch, a *warning* message is issued. Whenever the value in a location is used, its run-time type tag is checked, and if the type is inappropriate in the context in which the value is being used, an *error* message is issued.

The tool is able to find all of the run-time storage violations found by Purify (e.g., a use of an uninitialized variable or an out-of-bounds array access). In these cases, the tool's error messages are roughly equivalent to those reported by Purify on a given run of a faulty program. The warning messages, however, provide more information about what occurred prior to the error, which can be of great help when trying to identify the statements that actually caused the error. In addition, the tool has the potential to find errors that Purify cannot detect (e.g., a write into one member of a union followed by a read from a member of a different type).

In preliminary tests, the tool has been used to find bugs in several Solaris utilities and Olden benchmarks. The information provided by the tool is usually succinct and precise in showing the location of the error.

The remainder of the paper is organized as follows: Section 2 provides several examples that illustrate how the tool works and what kinds of errors it can detect; Section 3 describes a preliminary implementation of the tool; Section 4 discusses the results of some experiments; and Section 5 concerns related work.

2 Motivating Examples

In this section, we provide three motivating examples to illustrate the potential benefits of providing run-time type checking. In each case, we describe the kind of error that might be made, how our tool would detect the error at run-time, and the interesting issues raised by the example.

2.1 Bad Union Access

A very simple example of a logical error that manifests itself as a bad run-time type is writing into one field of a union and then reading from another field with a different type. This is illustrated by the following code fragment:

```
1. union U { int u1; int *u2; } u;
2. int *p;
3. u.u1 = 10; /* write into u.u1 */
4. p = u.u2;  /* read from u.u2 – warning! */
5. *p = 0;    /* bad pointer deref – error! */
```

In this example, an integer value is written into variable `u` (on line 3), and is subsequently read as a pointer (on line 4). The value that is read from `u` is stored in variable `p`, which is then dereferenced (on line 5). The symptom of the error is the attempt to use the value 10 as an address on line 5; however, the actual point of the error can be said to be on line 4, when a value of one type is read as if it were another type (i.e., the run-time type of `u.u2` is not the same as its static type).

A tool like Purify would report an error when line 5 was executed; however, it would not be able to point to line 4 as the source of the error.

Recall that our tool instruments the program to track the run-time types of memory locations. In the example, the single location that corresponds to both `u.u1` and `u.u2` would have an associated run-time type. That type would be set to `int` after the assignment `u.u1 = 10` on line 3. On line 4, the location is read, and its value is assigned to a pointer; this is a type mismatch, and therefore our tool would produce a warning message when line 4 is executed (as well as an error message reporting the run-time type violation at line 5).

2.2 Heterogeneous Arrays

C programmers sometimes try to avoid the overhead of the `malloc` and `free` functions by writing their own dynamic memory-management functions. For example, a programmer might allocate a large chunk of memory using a single call to `malloc` via an assignment like the following:

```
char *myMemory = (char *)malloc(BLOCKSIZE);
```

(where `BLOCKSIZE` is some large integer value). Subsequently, when new memory is needed, a call to a user-defined function, e.g., `myMalloc`, is made, rather than a call to `malloc`. The `myMalloc` function returns a pointer to an appropriate part of the `myMemory` "chunk". Similarly, calls to `free` are replaced by calls to `myFree`, which updates appropriate data structures to keep track of which parts of `myMemory` are currently in use.

The `myMemory` "chunk" could be used as a heterogeneous array; i.e., different parts of the array could contain values of different types. For example, the programmer's code might include the following declarations and calls:

```
1. struct node { int data; struct node *next; } *n, *tmp;
2. int *p = (int *) myMalloc(100 * sizeof(int));
3. n = (struct node *) myMalloc(sizeof(struct node));
```

The call on line 2 allocates an array of 100 integers, and the call on line 3 allocates one node for a linked list.

Now suppose that there is a bug in the programmer's memory-allocation code that causes it to return overlapping chunks of memory. In particular, assume that the value assigned to variable `n` on line 3 is the same as the address of `p[98]`. In addition, assume that pointers and integers both take 4 bytes, and that there is no padding between the two fields of `struct node`. In this case, after the call to

myMalloc on line 3, the address of n->data is the same as the address of p[98], and the address of n->next is the same as the address of p[99]. Now consider what happens when the following statements are executed:

```
4. n->next = (struct node *) myMalloc(sizeof(struct node));
5. p[99] = 0;
6. tmp = n->next;
```

Since p[99] and n->next refer to the same location, the assignment on line 5 overwrites the value assigned to n->next on line 4 with the value 0, essentially replacing the link to the next node in the list with a (list-terminating) NULL. Therefore, future accesses to the list will find only one node. If the assignments on lines 4 and 5 were in different parts of the code (e.g., in unrelated functions) the source of this error might be very difficult to track down (and a tool like Purify would not be able to help, since there are no bad pointer dereferences or array-access errors. Of course, if the assignment on line 5 set p[99] to some value other than zero, then future accesses to the list would probably cause a bad pointer dereference, which would be detected by a tool like Purify. However, as in the "bad union access" example above, Purify would not be able to locate the source of the error.)

Our tool would tag the elements of myMemory with their run-time types. For example, after the assignment on line 4, the location that corresponds to n->next would be tagged with type pointer. The assignment on line 5 would change that tag to int. Finally, the use of the value in n->next on line 6 would cause a warning message to be reported, because the location is tagged with run-time type int while its value is being assigned to a pointer (tmp).

2.3 Using Structures to Simulate Inheritance

C is not an object-oriented language, and therefore has no classes. However, programmers often try to simulate some of the features of classes using structures[4]. For example, the following declarations might be used to simulate the declaration of a superclass Sup and a subclass Sub:

```
struct Sup { int a1; int a2; };
struct Sub { int b1; int b2; char b3; };
```

A function might be written to perform some operation on objects of the superclass:

```
void f ( struct Sup *s ) {
   s->a1 = ...
   s->a2 = ...
}
```

and the function might be called with actual arguments either of type struct Sup * or struct Sub *:

```
struct Sup sup;
struct Sub sub;
f(&sup);
f(&sub);
```

The ANSI C standard guarantees that the first field of every structure is stored at offset 0, and that if two structures have a common initial sequence – an initial sequence of one or more fields with compatible types – then corresponding fields in that initial sequence are stored at the same offsets. Thus, in this example, fields `a1` and `b1` are both guaranteed to be at offset 0, and fields `a2` and `b2` are both guaranteed to be at the same offset. Therefore, while the second call, `f(&sub)`, would cause a compile-time warning (which could be averted with an appropriate type cast), it would cause neither a compile-time error nor a run-time error, and the assignments in function `f` would correctly set the values of `sub.b1` and `sub.b2`.

However, the programmer might forget the convention that `struct Sub` is supposed to be a subtype of `struct Sup`, and might change the type of one of the common fields, might add a new field to `struct Sup` without adding the same field to `struct Sub`, or might add a new field to `struct Sub` before field `b2`. For example, suppose the declaration of `struct Sub` is changed to:

```
struct Sub { int b1; float f1; int b2; char b3; };
```

Now, when the second call to `f` is executed, the assignment `s->a2 = ...` would write into the `f1` field of sub rather than into its `b2` field. The fact that the `b2` field is not correctly set by the call to `f`, or the fact that the `f1` field is overwritten with a garbage value will probably either lead to a run-time error later in the execution, or will cause the program to produce incorrect output.

Once again, the use of run-time types can help. The assignment `s->a2 = ...` causes `sub.f1` to be tagged with type `int`. A later use of `sub.f1` in a context that requires a `float` would result in an error message due to the mismatch between the required type (`float`) and the current run-time type (`int`).

Note that in this example, a tool like Purify would not report any errors, because there are no bad pointer or array accesses: function `f` is not writing outside the bounds of its structure parameter, it just happens to be the wrong part of that structure from the programmer's point of view.

3 Implementation

Our debugging tool has been implemented for all of ANSI C. It has two major components: a compiler front-end that instruments the program, and a run-time system that tracks the dynamic type associated with each memory location.

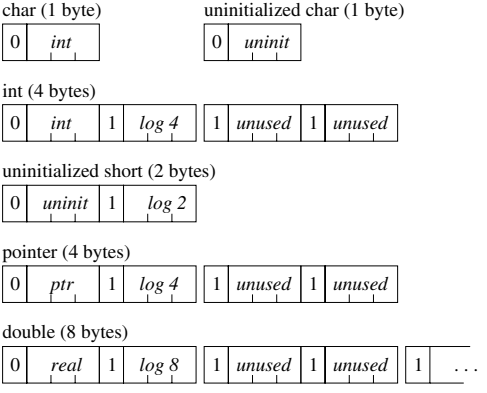
3.1 Tracking Type Information

The run-time types that are tracked are: *unallocated*, *uninitialized*, *integral*, *real*, and *pointer*. For aggregate objects (structures and arrays), each field/element has

its own tag. For pragmatic reasons, typedefs are ignored, and different pointer types are not distinguished. While it might be useful to distinguish different user-defined types and different kinds of pointers, doing so would mean having no *a priori* upper bound on the number of types in a program, and therefore no upper bound on the number of bits required to represent a type, which would greatly complicate run-time type checking.

The run-time component of our tool is implemented by storing type information in a "mirror" of the memory used by the program. Each byte of memory maps to a four-bit nibble in the mirror. Of these four bits, one "continuation" bit encodes the extent of the object (0 denotes the start of a new object, 1 denotes a "continuation" nibble), and three "data bits" encode other information. In the first nibble of an object's tag, the "data bits" encode the object's current type; in the second nibble (if the object is larger than one byte in size) the "data bits" encode (log_2 of) the size of the object. This scheme allows for quick comparisons between two objects by merely comparing the first eight bits (two *nibbles*) of their tags. For objects larger than two bytes, the remaining "data bits" are currently unused (they may potentially be used to encode information for future enhancements or optimizations).

The tags for some common scalar types (with their sizes) are illustrated below:



The mirror is allocated in segments corresponding to 1MB of user memory, as the amount of memory in use by the program increases. Pointers to these "mirror pages" are stored in a table indexed by the most significant 12 bits of the user-space address, so accesses to an object's tag are fast. The interface to the run-time system consists primarily of procedures (implemented using macros whenever possible to cut down on the overhead of function calls) that set a tag (`setUninitTag` and `setScalarTag`), copy a tag (`copyTag`), and verify that a tag agrees with an expected type (`verifyTag`). There is also a procedure (`verifyPtr`) to verify that a pointer points to allocated memory before it is dereferenced, and a set of procedures to handle the passing of function parameters and return values (`processArgTag` and `processReturn`).

3.2 Source-Level Instrumentation

To instrument a program, the tool performs a source-to-source transformation on the C source files using Ckit[5], a C front end written in ML. Working at the source level gives the tool access to all the source-level type information it needs. Also, the flexibility of the comma operator in C makes it possible to preserve the ANSI C semantics of the original program while retaining portability: an instrumented C file can, in principle, be compiled on multiple platforms.[1]

Handling the C language was non-trivial because a number of C's features make correct instrumentation difficult. The following summary of the instrumentation actions that the tool performs highlights some of the issues:

main: Every occurrence of `main` is renamed to `prog_main`. Our run-time system defines its own `main` function, which performs some initialization before calling `prog_main`. This way, we can filter out command-line arguments for the run-time system, and initialize the tags for the `argv` arrays. Also, this way recursive calls to `main` do not cause any problems.

statements and expressions: Each program statement and its component expressions are instrumented via a set of syntax-directed transformations. Code for setting, copying, or verifying tags is added to expressions; the instrumented code makes extensive use of the comma operator (see Section 3.3 for an example).

locals: Local variables are initially tagged *uninitialized*. A local variable that is initialized is processed as if the initialization expression were assigned to that variable. Because the instrumentation code needs to be able to take the address of all variables, `register` variables are demoted to `auto` variables. Also, tags for bit fields are not initialized because C does not allow the address of a bit field to be taken; instead, bit fields remain tagged as *uninitialized*. This still triggers type-violation warnings and errors if a bit field is used in a type-unsafe manner.

globals: Tags for global variables are initialized in a special `init` function; one such function is created per source file. Our `main` function calls each of these `init` functions before calling `prog_main`. The list of `init` functions from the different source files is collected at link time.

externs: `extern` variables that are not defined in any of the instrumented source files are treated specially. To allow instrumented source files to be linked with uninstrumented object code (most commonly library modules), we assume that `extern` variables are "well-behaved", and so initialize their tags to contain their declared types. However, the tool is limited by what is visible to it. In particular, it cannot initialize the tags for incomplete array types (e.g., `extern int a[];`) because the size of the array is not visible.

calls: To handle function calls, the tags of the function's parameters must be communicated between the caller and the callee. At the callsite, code is added to store the tags for the actual parameters in an array, whose address is kept in a global pointer, `globalArgTags`. At the head of the function definition, code is

[1] Note: the Ckit front end does not currently support C-preprocessor directives, so at present we can only instrument *preprocessed* C code. This limits portability to some extent, but is not a fundamental limitation of our approach.

added to extract the tags of the parameters passed to the function. The same mechanism is used to pass the tag of the return value back to the callsite. To allow a mix of instrumented and uninstrumented functions to work properly, including where instrumented functions are invoked via callbacks from uninstrumented functions, the instrumented caller stores the address of the callee in a global pointer, `globalCallTarget`, while the instrumented callee compares its own address with `globalCallTarget`. If the addresses match, it means that the caller is instrumented, so the tags for function arguments and return value are processed as described above. If the addresses do not match, however, it means that the caller is uninstrumented, so tags for function parameters cannot be extracted from `globalArgTags`.

return: At a return statement and at the end of a function f, the mirror for the entire stack frame of f must be cleared to *unallocated*. This is done by `processReturn`, a procedure in our run-time system. The start of the stack frame for f is assumed to be the greatest[2] of the addresses of f's formal parameters (if any) and the first local variable declared in f (a variable specially added by our instrumentation process). Since the call to `processReturn` itself has advanced the stack-frame pointer beyond the end of f's stack frame, a lower bound on the end of f's stack frame is obtained by taking the address of a local variable declared within `processReturn` itself.

3.3 Instrumentation Example

A C program is instrumented by transforming the statements contained in the bodies of the program's functions. Expressions in each statement are replaced by the result of a call to the instrumentation function *instr-expr*, which takes as arguments (1) *exp*, the expression to be instrumented, (2) a boolean flag *enforce*, and (3) an optional pointer argument *tagptr*.

The *enforce* flag specifies whether the expression's run-time type must match its static type, i.e., if the expression is used in a context that is sensitive to its type. For example, in the expression $e_1 + e_2$, the data in both e_1 and e_2 need to reflect their respective static types, so *enforce* must be *true* when instrumenting both e_1 and e_2. On the other hand, for the expression `&`e, the type of the data in e is not relevant to the type-safety of `&`e, so e would be instrumented with *enforce=false*.

The presence of the optional pointer argument *tagptr* indicates, when instrumenting an expression e, that an enclosing expression needs access to e's type. In such a case, *tagptr* is used to convey this information.

The output of applying *instr-expr* to three common expressions ($e_1 = e_2$, *id*, and $*e$), is shown in Table 1.[3]

[2] Assuming that the stack grows downwards in memory (from high to low addresses). For stacks that grow upwards, we use the lowest of the addresses of f's formal parameters and its first local variable.

[3] We omit some details that would only complicate the example. For instance, we actually perform slightly different actions for instrumenting lvalues and rvalues. Also,

Table 1. Examples of instrumentation rules

exp	***instr-expr*(*exp*,*enforce*)**	***instr-expr*(*exp*,*enforce*,*tagptr*)**
id	*(verifyTag(&*id*, *typeof*(*id*)),[1] &*id*)	*(*tagptr* = &id, verifyTag(&*id*, *typeof*(*id*)),[1] &*id*)
$*e$	*(tmp$_{ptr}$ = *instr-expr*(*e*, *true*), verifyTag(tmp$_{ptr}$, *typeof*($*e$)),[2] tmp$_{ptr}$)	*(*tagptr* = *instr-expr*(*e*, *true*), verifyTag(*tagptr*, *typeof*($*e$)),[2] *tagptr*)
$e_1 = e_2$	(tmp$_{assign}$ = *instr-expr*(e_1, *false*, tmp$_{ptr1}$) = *instr-expr*(e_2, *enforce*, tmp$_{ptr2}$), copyTag(tmp$_{ptr1}$, tmp$_{ptr2}$, *typeof*(e_1)), tmp$_{assign}$)	(tmp$_{assign}$ = *instr-expr*(e_1, *false*, *tagptr*) = *instr-expr*(e_2, *enforce*, tmp$_{ptr2}$), copyTag(*tagptr*, tmp$_{ptr2}$, *typeof*(e_1)), tmp$_{assign}$)

[1] omit if *enforce* = *false*
[2] call verifyPtr instead if *enforce* = *false*

The tmp variables that appear in the rules in Table 1 are temporaries (of appropriate type) introduced by the instrumentation code. The instrumented expression is shown in the second and third columns: column two shows how instrumentation is carried out when the optional third argument is absent; column three shows the instrumentation strategy when the third argument, *tagptr*, is present. At run-time, the *tagptr* variable will be set to point to an object whose mirror is tagged with the expression's dynamic type, to be used in the instrumentation code of an enclosing expression (see the rules for $e_1 = e_2$). When *enforce* is *true*, the verifyTag procedure is used to verify that the tag associated with a given object agrees with its declared type.

For the *id* case, the only check done (when *enforce* = *true*) is to verify that *id*'s dynamic type agrees with its declared type.

For the dereference case, the subexpression e is first instrumented by passing *enforce* = *true* to *instr-expr* (since e will be dereferenced, i.e., used as a pointer). After that, if *enforce* = *true*, we verify that the dynamic type of $*e$ agrees with its declared type. If *enforce* = *false*, we do not require that $*e$'s dynamic type match its declared type; however, we still want to make sure that $*e$ is not *unallocated* (i.e., that e points to valid memory). This is performed by the verifyPtr procedure, which allows the tool to output an error message before an invalid pointer dereference occurs.

In the assignment case, expression e_1 is instrumented with *enforce* = *false*, since we do not care about the type of the data that is about to be overwritten; e_2 is instrumented with *enforce* = *true* only if the assignment expression is being instrumented with *enforce* = *true*. The copyTag procedure copies the tag of the right-hand-side expression to the mirror of the left-hand-side expression, and

if an expression's lvalue or rvalue is not used in a larger context, we avoid preserving that value in the instrumented expression.

also issues a warning message if the type of the right-hand-side expression is not compatible with the static type of the left-hand-side expression.

For the *id* and $*e$ cases, the instrumented code has the form `*(...,`*ptr*`)`; this is so that the instrumented expression is a valid lvalue. An assignment expression is not an lvalue, and so does not need to be instrumented in this way. However, if an assignment is used as an rvalue in a larger context, we must still make sure that the instrumented expression preserves the correct rvalue, which is the purpose of $\texttt{tmp}_{assign}$.

Now consider instrumenting the statement `x = *p;`. Since this assignment does not occur in a context that uses its value, its type need not be verified, and the assignment is instrumented with *enforce* = *false*. Also, since there is no enclosing expression that needs access to the assignment's type, the optional *tagptr* argument is not needed. Assuming `x` is of type `int` and `p` is of type `int *`, the result of *instr-expr*(`x = *p`, *false*) is shown in Figure 1. Notice that the `tmp` variable at the outermost level (for the assignment expression) has been omitted because in this case, the assignment's rvalue does not need to be preserved.

```
*(tmp1 = &x, &x) =
   *(tmp2 = *(verifyTag(&p, pointer_type),
              &p),
     verifyPtr(tmp2, sizeof(int)),
     tmp2),
copyTag(tmp1, tmp2, int_type)
```

Fig. 1. Output of *instr-expr*(`x = *p`, *false*)

3.4 Other Features

In order to perform proper type checking, the tool needs to handle memory-management functions specially. We replace each call to `malloc` (and its relatives) with our own version that, upon successfully allocating a block of memory, initializes the mirror for that memory block with the *uninitialized* tag. Similarly, our `free` function resets the mirror to be of *unallocated* type. Our versions of these functions do their own bookkeeping so we know how many bytes are being freed by a call on `free` at run-time. Our version of `malloc` also adds padding between allocated blocks to decrease the likelihood of a stray pointer jumping from one block to another (this is the approach used by Purify). For the stack allocation function `alloca`, we instrument the callsites to additionally invoke procedures to initialize the newly allocated stack space.

Another routine that is handled specially is the variable argument routine `va_arg` (usually implemented as a macro). Our portable solution assumes that the argument obtained is properly typed, so we instrument each invocation of `va_arg` to return a tag of the expression's static type.

As mentioned in Section 3.2, the approach we have taken allows us to link instrumented modules with uninstrumented ones, with the only requirement being that the program's `main` function must be renamed to `prog_main`. This flexibility is useful if, for example, a programmer only wants to debug one small component of a large program: they can instrument just the files of interest, and link them in with the other uninstrumented object modules. A caveat when doing this, however, is that it may lead to the reporting of spurious warning and error messages because the uninstrumented parts of the code do not maintain type information. For example, if a reference to a valid object in the uninstrumented portion of the program is passed to an instrumented function, the tool will think that the object is unallocated, and may output spurious errors and warnings.

This problem extends, in general, to library modules. For example, the flow of values in a function like `memcpy`, the initialization of values from input in a function like `fgets`, and the types of the data in a static buffer returned by a function like `ctime` would not be captured. To handle these, we have created a collection of instrumented versions of common library functions that affect type flow. These are wrappers of the original functions, hand-written to perform the necessary tag-update operations to capture their type behavior.

Finally, our tool lends itself naturally to interactive debugging. When a warning or error message is issued, a signal (`SIGUSR1`) is sent, and can be intercepted by an interactive debugger like GDB[6]. The user is then able to examine memory locations, including the mirror, and make use of GDB's features to better track down the cause of an error.

4 Experiments

To test the effectiveness of our debugging tool, we used Fuzz[7] to find Solaris utilities that crash on some inputs, and instrumented five such programs for testing (`nroff`, `plot`, `ul`, `units`, `col`). We also tracked down bugs that appear in two programs from the Olden benchmark suite (`health`, `voronoi`). A summary of what our tool revealed about these runs is given below.

nroff: An array of pointers is accessed with a negative index, and the retrieved word, when dereferenced, causes a segmentation fault. The instrumented program, before crashing, warns that the retrieved word that is about to be dereferenced actually contains an array of characters.

plot: A rogue pointer, after passing beyond the bounds of a local array, walks up the stack, writing bytes as it goes. It eventually attempts to write to invalid memory, at which point the program crashes. The instrumented program outputs a long list of warning messages signaling these writes to unallocated memory, accurately identifying the line of code where this occurs.

ul: The original program crashes during a call to `fgetwc`, while the instrumented program crashes during a call to `fprintf` as our instrumentation code is attempting to write a warning message. The cause of the crash in the original program was difficult to diagnose, but "accessing unallocated memory" error messages generated by our instrumented program led us to the cause of the crash:

a pointer, after passing beyond the bounds of an array, walks through the bss section and eventually overwrites part of the global `_iob` array (which contains information about `stdin`, `stdout`, and `stderr`). This causes the subsequent call to `fgetwc` in the original code, and `fprintf` in the instrumented code, to crash.

units: In the original program, an errant pointer manages to corrupt the "save" area of the call stack, resulting in bizarre behavior that was difficult to track down without our tool. The instrumented program issues a type-violation error message after the character pointer `cp` is set to point to itself, and is subsequently used to write a character value onto itself. The next dereference of `cp` generates another error message, and then the program crashes.

col: The original program crashes on a dereference of a bad pointer, but our instrumented program does not crash; instead, it fails to terminate (at least, after two hours we stopped waiting for it to terminate). The first of many error messages generated by our instrumented program signals a dereference into unallocated memory, and points to the line in the program where the crash occurs (in the uninstrumented code). The point where the error message was generated is close to where the pointer first stepped out of bounds of the global array to which it pointed.

health: In the semantics of C, memory allocated by `malloc`, unlike `calloc`, is not required to be zero-initialized, although many programmers assume that it is. In this program, the pointer fields in two recursive data structures are not initialized after allocation via `malloc`. While traversing these structures, the original program counts on the pointer fields being `NULL` to indicate the absence of a substructure. The instrumented program warns of an access to uninitialized memory each time the program checks to see if one of these pointer fields is `NULL`. On our testing platform, all the memory allocated with `malloc` in this program happen to be zero-initialized and so the program does not crash. However, the erroneous assumption about `malloc` is a program flaw that may cause a crash on a different execution (or when the program is run on a different platform).

voronoi: Some bit-level manipulations are performed on a pointer to a struct, yielding a pointer to a "field" that does not belong to the struct, since some assumptions made by `voronoi` about the size of the struct do not hold on our test machine. A subsequent assignment of this pointer (as a function argument) generates a warning message stating that an unallocated object is being passed. Later, when the pointer (which happens to be `NULL`) is actually dereferenced, the instrumented program gives an "accessing unallocated memory" error message before crashing.

In most cases, crashes in the test programs were found to have been caused by a pointer (or array index) that had gone astray. In every case, our tool was able to detect the out-of-bounds memory accesses because the type of the pointed-to memory was different from the expected type. While these results are very encouraging, these kinds of errors would also be detected by Purify.

We can easily create examples (such as the ones given in Section 2) for which our tool is able to detect errors that are *not* detected by Purify; however, we have not yet found examples of those kinds of bugs in real programs. We suspect that

Table 2. Performance on the benchmarks ("Lines of C code" reports the number of unpreprocessed lines of source code, with comments and blank lines removed.)

source	program	lines of C code	running time (secs) uninstrumented	instrumented	slowdown
Olden benchmarks	bh	1,049	8.97	910.02	101.4
	bisort	570	7.60	70.86	9.3
	em3d	414	1.79	31.35	17.5
	health	559	6.45	56.97	8.8
	mst	493	3.25	83.10	25.6
	perimeter	389	2.22	49.13	22.1
	power	679	6.41	241.83	37.7
	treeadd	291	3.90	62.17	15.9
	tsp	567	12.78	83.64	6.5
SPEC benchmarks	compress	1,491	19.87	695.83	35.0
	gcc	151,531	11.08	1288.64	116.3
	go	26,917	12.04	654.86	54.4
	li	6,272	5.47	320.99	58.7
	m88ksim	14,502	1.80	239.91	133.0
	vortex	52,624	12.37	1596.02	129.1
Solaris utilities	col	502	1.47	29.39	20.0
	nroff	11,018	0.82	53.01	64.6
	plot	326	1.02	5.90	5.8
	ul	468	0.33	1.87	5.6
	units	457	0.39	3.19	8.2

such bugs are more likely to occur in larger, more complicated programs, but due to limitations of the current version of the Ckit front end, we have not been able to successfully compile many large programs. Furthermore, the code that we have used to date for testing our technique is in most cases robust code that has been in use for quite some time. As a result, the likelihood of finding errors is lower than if the tool were applied to code during the software-development cycle.

Not surprisingly, the extensive checking performed by our tool comes at a performance cost. This cost is due to the execution of our type-tracking procedures, as well as to the transformation of the original program's expressions into more complicated ones in order to allow type tracking while preserving the original expressions' values, types, and side-effects. To measure the execution-time overhead that is introduced by our tool, we instrumented the five Solaris utilities described above, as well as several programs from the SPEC and Olden benchmarks. The benchmarks were optimized with gcc's -O2 option,[4] and executed with legitimate (non-crashing) inputs on a 300 MHz Sun Ultra 10 work-

[4] Except for `gcc`, for which the optimized instrumented version behaved differently from the original version, for unknown reasons.

station with 256 MB of RAM and 1.1 GB of virtual memory. The sizes of the benchmarks, as well as the execution times (user+system time, in seconds) and slowdowns are reported in Table 2.

The slowdowns we observe on these benchmarks range from about 6 times to 130 times, with a median of 23.8. As a point of comparison, the slowdown factor for Purify tends to be in the range of 10 to 20. The exorbitant slowdown for `bh` is due mainly to the program making many assignments of structures, for which our tag-copying procedure `copyTag` performs an inefficient nibble-by-nibble copy. For the SPEC benchmarks, as well as for `nroff`, the performance degradation is largely due to the overhead of writing out spurious warning and error messages. These mainly result from the tool's inability to cleanly capture the type behavior of `calloc`-initialized memory and incomplete array types (in particular, the `ctype` array, which is used by `ctype.h` macros), and also from some of these programs performing masking operations which treat integers as arrays of characters – technically a type violation.

Future work includes using the results of static analysis to reduce the amount of instrumentation introduced by our tool (thus reducing its overhead). For example, if the value in a location is used multiple times, and there is no possibility that its type is modified between the uses, then only the first use needs to be checked.

5 Related Work

Run-time type-checking is not a new idea. It has been implemented in functional-style languages like LISP and Scheme, and object-oriented languages like Java, Smalltalk and Objective-C. However, designing dynamic type-checking for a language like C that includes unions, structures, arrays without bounds checking, casting and pointer arithmetic is a large undertaking. Additionally, there is a subtle difference in approach. Traditional dynamic type-checking attaches tags to data, while our approach separates the tag space from the data space. Both approaches attain good spatial locality of accesses (in the case of separated tags and data, the locality of tag accesses mirrors the locality of data accesses). However, if checking of some of the tags can be eliminated based on static analysis, the locality of accesses to data does not suffer as it does in the case of co-located tags and data.

Another significant advantage is the fact that the tags can be protected from erroneous accesses by the user code. User code is simply less likely to make erroneous accesses to the separated tag space, and if better guarantees are required, memory-protecting hardware can be used to shield the tag space during the execution of user code.

The concept of soft typing [8,9] was introduced in an effort to combine the benefits of static and dynamic typing for functional-style languages. In this approach, static typing is employed to identify program statements that do not statically type check. These statements are subsequently instrumented to be dynamically type-checked. The earlier work [8] concentrates on presenting a framework for soft typing on a restricted class of programming languages; the

latter work [9] extends this work to handle realistic languages like Scheme. By addressing a traditionally dynamically typed language, the emphasis was put on the application of soft typing as a way of lowering the run-time overhead of type-checking. While this body of work is similar to ours in its approach to type-checking, the differences in the languages handled by the two approaches is significant.

Approaches to detection of errors in C programs by means of executing a program instrumented to perform run-time checks have been developed in the past. Safe-C[2] provides run-time detection of array access and pointer dereference errors, such as array out-of-bounds errors, stale-pointer accesses, and accesses resulting from erroneous pointer arithmetic. This is done by keeping track of attributes of the referent of each pointer by transforming C code to C++ code, and taking advantage of operator overloading to perform appropriate checks whenever certain operators are applied. Purify[1] detects errors similar to those found by Safe-C, and, in addition, identifies uninitialized memory reads and memory leaks. Purify performs these checks by instrumenting object files and modifying the layout of heap-allocated memory in order to catch access errors. Our approach catches all of these errors in addition to run-time type violations that are not covered by Purify and Safe-C. Furthermore, the warning messages provided by our tool provide a history of suspicious type propagation that can aid in pinpointing the true cause of an error.

In the realm of security, tools have been developed to prevent "stack smashing" (where the return address in the activation record is modified by a malicious agent to obtain control of the program)[10,11]. Our tool also detects such attacks, which fall under the general category of "type errors" detected by our tool.

A technique to enable efficient checking of array-access and pointer-dereference errors in a multiprocessor environment was presented in [3]. They achieve low-cost checking by creating a version of the program that contains only computations that affect pointer and array accesses, instrumenting that version, and running it in parallel with the original program. We may be able to use this technique to improve our tool's performance.

There have also been a number of efforts to address the problem of identifying errors in C programs due to out-of-bounds array indexes and misuses of type casts based on the use of static analysis. Work on static analysis that can be applied to checking for out-of-bounds array accesses includes [12,13,14,15,16]. Algorithms for points-to analysis that distinguish among fields of structures [17, 18] and for so-called "physical type checking" [19] can also be used to perform static safety checks. However, most of the work based on static analysis cited above has used flow-insensitive techniques, which is likely to cause an enormous number of warnings of possible misuses to be generated when applied to safety checking of real-life C programs. The advantage of a dynamic type-checking tool like the one reported in this paper is the ability to obtain more accurate information about type misuses and access errors, albeit only for ones that occur during a given run of the program.

References

1. R. Hasting and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter Usenix Conference*, 1992.
2. T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, 1994.
3. H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software–Practice and Experience*, 27(27):87–110, 1997.
4. M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. In *Proc. of ESEC/FSE '99: Seventh European Softw. Eng. Conf. and Seventh ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, pages 180–198, September 1999.
5. Ckit. http://cm.bell-labs.com/cm/cs/what/ smlnj/doc/ckit/.
6. R. Stallman and R. Pesch. *Using GDB: A Guide to the GNU Source-Level Debugger*. July 1991.
7. B. Miller, D. Koski, C.P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin-Madison, 1995.
8. M. Fagan. Soft typing: An approach to type checking for dynamically typed languages. Technical Report TR92-184, Department of Comp. Sci., Rice Univ., Houston, TX, USA, March 1998.
9. A. Wright. Practical soft typing. Technical Report TR94-236, Department of Comp. Sci., Rice Univ., Houston, TX, USA, April 1998.
10. Immunix stack guard. http://immunix.org/stackguard.html.
11. Stack shield. http://www.angelfire.com/sk/stackshield/info.html.
12. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conf. Rec. of the Fifth annual ACM Symp. on Princ. of Prog. Lang.*, pages 84–96. ACM, January 1978.
13. C. Verbrugge, P. Co, and L.J. Hendren. Generalized constant propagation: A study in C. In *6th Int. Conf. on Compiler Construction*, volume 1060 of *Lec. Notes in Comp. Sci.*, pages 74–90. Springer, April 1996.
14. R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 182–195, New York, NY, 2000. ACM Press.
15. R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating array bounds checks on demand. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 321–333, New York, NY, 2000. ACM Press.
16. D. Wagner, J.S. Foster, E.A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Symposium on Network and Distributed Systems Security (NDSS '00)*, pages 3–17, San Diego, CA, February 2000.
17. B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *6th Int. Conf. on Compiler Construction*, volume 1060 of *Lec. Notes in Comp. Sci.*, pages 136–150. Springer, April 1996.
18. S. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 91–103, May 1999.
19. S. Chandra and T. Reps. Physical type checking for C. In *Proc. of PASTE '99: SIGPLAN-SIGSOFT Workshop on Program Analysis for Softw. Tools and Eng.*, pages 66–75, New York, NY, 1999. ACM.

Library-Based Design and Consistency Checking of System-Level Industrial Test Cases

Oliver Niese[1], Bernhard Steffen[2], Tiziana Margaria[1], Andreas Hagerer[1], Georg Brune[3], and Hans-Dieter Ide[3]

[1] METAFrame Technologies GmbH, Dortmund, Germany
{ONiese, TMargaria, AHagerer}@METAFrame.de
[2] Chair of Programming Systems, University of Dortmund, Germany
Steffen@cs.uni-dortmund.de
[3] Siemens AG, Witten
{Georg.Brune, Hans-Dieter.Ide}@wit.siemens.de

Abstract. In this paper we present a new coarse grain approach to automated integrated (functional) testing, which combines three paradigms: *library-based test design*, meaning construction of test graphs by combination of test case components on a *coarse* granular level, *incremental formalization*, through successive enrichment of a special-purpose environment for application-specific test development and execution, and *library-based consistency checking*, allowing continuous verification of application- and aspect-specific properties by means of model checking. These features and their impact for the test process and the test engineers are illustrated along an industrial application: an automated integrated testing environment for CTI-Systems.

1 Introduction

The increasing complexity of today's testing scenarios for telephony systems demands for an integrated, open and flexible approach to support the management of the overall test process, i.e. specification of tests, execution of tests and analysis of test results. Furthermore, systems under test (*SUT*) become composite (e.g. including *Computer Telephony Integrated* (CTI) platform aspects), embedded (due to hardware/software codesign practices), reactive, and run on distributed architectures (e.g. client/server architectures). As a consequence, it becomes increasingly unrealistic to restrict the consideration of the testing activities to single units of the systems, since complex subsystems affect each other and require scalable, integrated test methodologies.

The requirements discussed in this paper exceed the capabilities of today's testing tools. To our knowledge there exist neither commercial nor academic tools providing comprehensive support for the whole system-level test process. In particular, most research on test automation for telecommunication systems concentrates on the generation of test cases and test suites on the basis of a formal model of the system: academic tools, like *TORX* [22], *TGV* [2], *Autolink* [13], and commercial ones like *Telelogic Tau* [21] presuppose the existence of

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 233–248, 2001.

fine-granular system models in terms of either automata or SDL descriptions, and aim at supporting the generation of corresponding test cases and test suites. This approach failed to enter practice in the scenario we are considering here, because it did not fit the current test design practice, in particular because there does not exist any fine granular formal model of the involved systems.

Therefore in our approach we aim at a formal methods-controlled, component-based test design on top of a library of elementary but intuitively understandable test case fragments. This establishes a coarse-granular 'meta-level' on which

- test engineers are used to think,
- test cases and test suites can be easily composed,
- test scenarios can be configured and initialized,
- critical consistency requirements including version compatibility and frame conditions for executability are easily *formulated* (see Section 5), and
- consistency is fully automatically enforced via *model checking* and error diagnosis.

Already after a few months of cooperation this coarse-granular test management support was successfully put into practice, drastically strengthening the pre-existing test environment. We are not aware of any other test environment systematically addressing the needs of coordinating the highly heterogeneous test process, let alone on the basis of formal methods.

The paper is organized as follows: Sect. 2 describes our application domain, system level testing of telephony systems, presents a concrete scenario, and introduces the requirements to the corresponding integrated testing environment. Sect. 3 presents our test coordinator tool, Sect. 4 describes the formal foundations of its design and analysis core, Sect. 5 shows the aspect-oriented character of the automated verification by discussing concrete classes of constraints, and Sect. 6 discusses the impact of the new environment on the test development practice. Finally Sect. 7 draws our conclusions.

2 System-Level Testing of Telephony Systems

As a typical example of an integrated CTI platform, Fig. 1 shows a midrange telephone switch and its environment. The switch is connected to the ISDN telephone network or, more generally, to the public service telephone network (PSTN), and acts as a "normal" telephone switch to the phones. Additionally, it communicates directly via a LAN or indirectly via an application server with CTI applications that are executed on PCs. Like the phones, CTI applications are active components: they may stimulate the switch (e.g. initiate calls), and they react to stimuli sent by the switch (e.g. notify incoming calls). In a system level test it is therefore necessary to investigate the interaction between such subsystems.

Typically, each participating subsystem requires an individual test tool (see Sect. 3). Thus in order to test systems composed of several independent subsystems that communicate with each other, one must be able to coordinate a

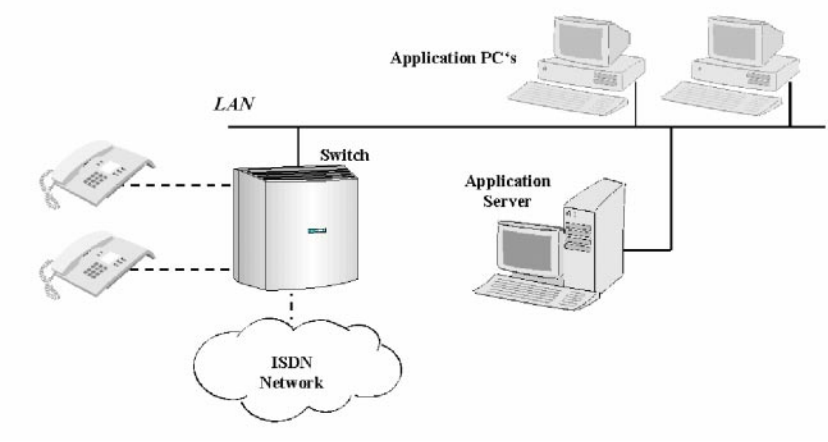


Fig. 1. Example of an Integrated CTI Platform

heterogeneous set of test tools in a context of heterogenous platforms. This task exceeds the capabilities of today's commercial test management tools, which typically cover the needs of specific subsystems and of their immediate periphery.

The remainder of this section explains and structures the corresponding central **requirements** for practice-oriented test management along the typical corresponding lifecycle: test case design, test organization, and test coordination.

2.1 Test Design Requirements

The design of test cases, i.e., specifying which control or inspection activities must be performed on the *SUT* and in which order, should neither require programming skills nor any knowledge of how to apply/use a specific test tool. In particular, concrete requirements concern the following areas:

Definition: Test cases should be graphically specifiable at the level of *SUT*-usage. A generalization by means of parameters should be supported.
Reuse: A macro mechanism should support the reuse of (partial) test cases. This automatically supports a hierarchical design style.
Validation: Consistency checks of tests cases at design time should guide the designer towards building (only) plausibile test cases.
Variation: Rule-based controlled as well as randomized parameter variation should enhance the expressiveness of the test results.

2.2 Test Organization Requirements

The central organizational aspects of the test process are:

Version control: (at the physical persistency level) Beside the test cases themselves, many other files referenced and used in test cases have to be organized, e.g. configuration files and test documentation. All these files evolve

throughout the test process. Therefore, it is important to capture the history of changes and the dependencies between versions.

Configuration management: It is mandatory, especially when considering integrated tests, that the *SUT* is in a well defined state before a test run is started. This is a non-trivial task because we treat complex systems, where the initialization is a distributed problem, and the initialization of one component typically affects the state of the others.

Structuring of tests: Tests must be structured to

- provide a simple mechanism to build test suites from test cases according to a variety of criteria, e.g. *regression test* or *feature test* for a certain test scenario.
- eliminate redundant test cases. This may dramatically reduce the whole test execution time, which is important for the scenario of Fig. 1, where new versions of the switch software must continuously be validated against the CTI applications.

2.3 Test Coordination Requirements

The whole test execution process must be supported, including:

Initialization: *SUT* components and test tools must be set into a well-defined starting state. Fast reinitialisation in case of repetition of a test case must be possible.

Execution: Distributed executed test tools of different abilities and different interconnection variants must be controlled in a way that emphasizes on the aspects control of tool activities and determination of state and state changes of *SUT* components for verification purposes. Reactions of *SUT* components on stimuli must be retrieved and evaluated. The evaluation results shall control succeeding test case steps. Timing constraints must be taken into account when stimulating a *SUT* component.

Reporting: Reporting shall record a test run and shall facilitate documentation and tracking of defects by providing sufficent details. A characterization of the *SUT*, i.e., versions of *SUT* components and of test tools, must be documented. Result and data of each step of the test case must be logged. The status of a test run must be summarized.

The next section describes how we attacked the requirements according to the test coordination aspect: design and execution of functional tests. Other aspects, like a sophisticated configuration management or the structuring of tests, are left to a subsequent phase.

3 The Test Coordinator

The realization of an adequate management layer for an automated test environment was attacked according to pragmatic criteria, of vital importance in an

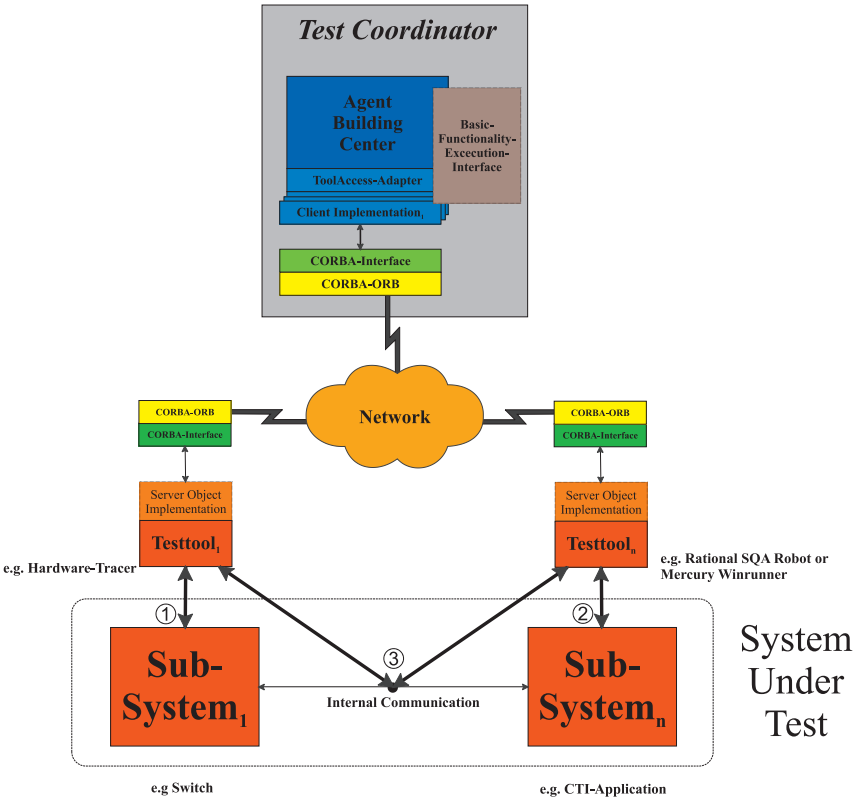


Fig. 2. Architectural Overview of the Test Environment

industrial development environment: first of all, the test environment should offer a viable execution environment (*Test Coordination Requirements*), then scale up to the required complexity (*Test Organization Requirements*), and also ease the test design phase (*Test Design Requirements*).

Accordingly, we built on an existing general purpose environment for the management of complex workflows, (METAFrame Technologies' *Agent Building Center* (ABC)) [18], which already encompasses most of the above mentioned features in an application-independent way. This way we were able to demonstrate in a short time the practical satisfiability of the kernel requirements concerning test coordination and test organization. In fact the currently available *Test Coordinator* (Fig. 2), which constitutes the test management layer of our environment, includes a specialization of the ABC for this application domain, i.e. system level testing of telephony systems. Meanwhile, the test management has already proved to be capable of coordinating the different control and inspection activities of integrated system-level tests.

Figure 2 shows the general architecture of the *Test Coordinator*. The *SUT* is composed of several subsystems, e.g. a telephone switch in cooperation with a

CTI application. Each subsystem is controlled via its own test tools[1]. The test tools have access to external interfaces (①, ②) as well as to some internal ones (③). They are steered by the Test Coordinator. For more details concerning the integration process of the test tools see [8].

Up to now, the effort in the project was mainly devoted to the support of test design and to the handling of advanced coordination and organization requirements.

3.1 ABC's Enabling Characteristics

The following characteristics of the ABC proved to be of major importance, due to the heterogeneous composition of the team (researchers, developers, and prospective industrial users coming from different groups within Siemens, with different focus on the project).

Behaviour-Oriented Development: In general, application development in the ABC, which goes far beyond the domain of CTI applications [18,17], consists of behaviour-oriented combination of building blocks on a *coarse* granular level. Building blocks are identified on a functional basis, understandable to application experts, and usually encompass a number of 'classical' programming units (be they procedures, classes, modules, or functions). They are organized in application-specific collections (palettes). In contrast to (other) component-based approaches, e.g., for object-oriented program development, ABC focusses on the dynamic behaviour: (complex) functionalities are graphically stuck together to yield flow graph-like structures embodying the application behaviour in terms of control. This graph structure is independent of the paradigm of the underlying programming language. In particular, we view this flow-graph structure as a control-oriented coordination layer on top of data-oriented communication mechanisms enforced, e.g., via RMI, CORBA or (D)COM. Concretely, the test management layer communicates with individual test tools by means of CORBA [10]. Accordingly, the purely graphical combination of building blocks' behaviours happens at a more abstract level.

Incremental Formalization: The successive enrichment of the application-specific development environment is two-dimensional. Beside the library of application-specific building blocks, which dynamically grows whenever new functionalities are made available, ABC supports the dynamic growth of a hierarchically organized library of *constraints*, controlling and governing the adequate use of these building blocks within application programs. This library is intended to grow with the experience gained while using the environment, e.g., detected errors, strengthened policies, and new building blocks may directly impose the addition of constraints. It is the possible *looseness* of these constraints which makes the

[1] Test tools can be e.g. proprietary hardware tracer for testing the switch or GUI test tools such as *Rational SQA Robot* [12] or *Mercury Winrunner* [7] for the test of applications.

constraints highly reusable and intuitively understandable. Here we consciously privilege understandability and practicality of the specification mechanisms over their completeness.

Library-Based Consistency Checking: Throughout the behaviour-oriented development process, ABC offers access to mechanisms for the verification of libraries of constraints via model checking. The model checker individually checks hundreds of typically very small and application- and purpose-specific constraints over the flow graph structure. This allows concise and comprehensible diagnostic information in the case of a constraint violation, in particular as the information is given at the application rather than at the programming level.

Taken together, these characteristics of the basic tool were actually the real enablers for the project results, in particular, as they provided a means to seamlessly coordinate the cooperation between ABC team, CTI expert, test designers, and test engineers.

4 Domain Modelling

This section summarizes the effort for instantiating the ABC as required for the CTI application. This mainly consists of the design of some application-specific building blocks, and the formulation of the frame conditions which must be enforced during test case design and test suite design.

4.1 Test Building Blocks

The ABC team, the CTI experts and the test designers define and also classify the building blocks occurring in the testing experiments, typically according to technical criteria like version or specific hardware or software requirements, origin (where they were developed) and, here, most importantly, according to their intent for the specific application area.

The resulting classification scheme is a simplified version of the taxonomies used in [6] and [17]. At this stage, the building blocks occurring in the test cases are organized in palettes, accessible from the *Test Coordinator's* GUI (see Fig.3, left), which are the basis for the test designers to graphically construct test cases by drag-and-drop. The test case design is completed by

- connecting the test blocks by edges, and
- configuring the internal parameters.

The resulting test graphs are directly executable for test purposes, and, at the same time, they constitute the models for our verification machinery by means of model checking. Figure 3 shows a typical test graph for illustration.

The palettes are thus central to build the *models*, and they constitute a vocabulary for the *constraint* definition in terms of modal formulas. Typically, the design of the classification scheme and of the constraints goes hand in hand with the definition of *aspect-specific views and filters*: both are mutually supportive means to an application specific structuring of the design process.

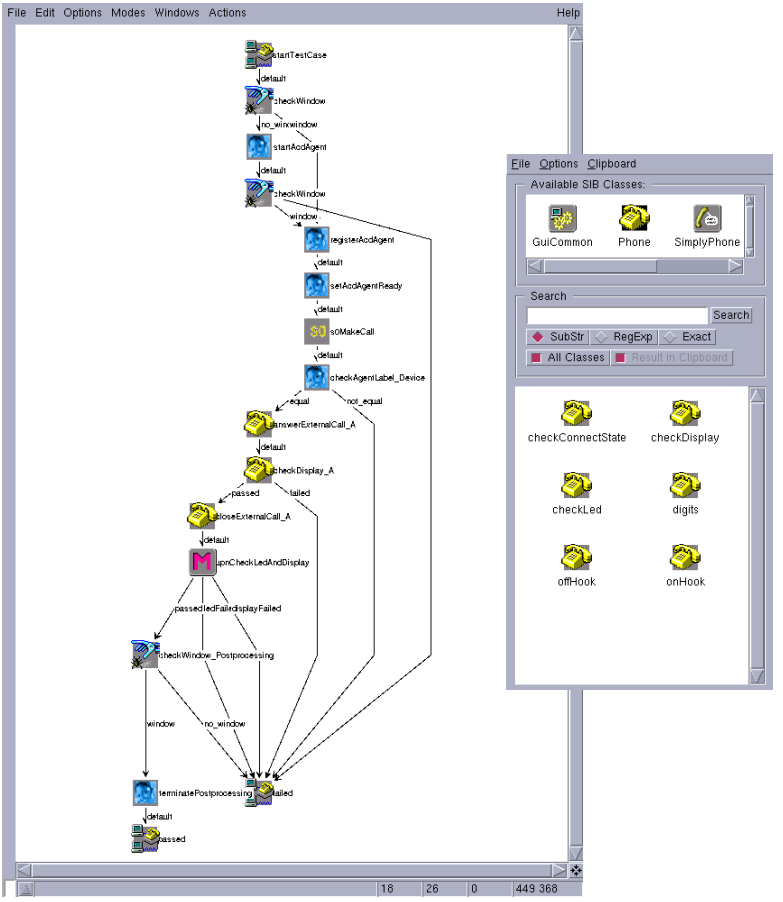


Fig. 3. A Test Graph and the Test Block Palettes

4.2 Models and Constraints

Formally, test graphs are modelled as flow graphs (see Fig. 3), whose nodes represent test blocks governing the stimulation of the SUT, and whose edges represent branching conditions steering the flow of control.

Definition 1 (Test Model).
A test model *is defined as a quadruple* $(\mathcal{S}, Act, \rightarrow, s_0)$ *where*

- $\mathcal{S}$ *is the set of available test blocks,*
- Act *is the set of possible branching conditions,*
- $\rightarrow \subseteq \mathcal{S} \times Act \times \mathcal{S}$ *is a set of transitions,*
- s_0 *is the uniquely determined initial test block.*

In ABC test models are subject to *local* and *global constraints* which, in conjunction, offer a means to identify 'critical' patterns in the test graph already

during the early design phase. The classification of constraints into local and global is important, since each kind requires a specific treatment. The on-line verification during the design of a new test, however, captures both kinds of constraints.

Local Constraints. Local constraints specify single test blocks in the context of their immediate neighbours: they capture a test block's branching potential as well as its admissible subsequent parameterization. Their correctness is checked by means of specific algorithms, which can be activated at need. The verification of local constraints is invoked automatically during the verification process of global properties.

Global Constraints: The Temporal Aspect. Global constraints allow users to specify causality, eventuality and other relationships between test blocks, which may be necessary in order to guarantee frame conditions for e.g., executability and version compatibility.

A test property is global if it does not only involve the immediate neighbourhood of a test block in the test model[2], but also relations between test blocks which may be arbitrarily distant and separated by arbitrarily heterogeneous submodels (see Section 5 for concrete examples).

The treatment of global properties is required in order to capture the essence of the expertise of designers about do's and don'ts of test creation, e.g. which test blocks are incompatible, or which can or cannot occur before/after some other test blocks. Such properties are rarely straightforward, sometimes they are documented as exceptions in thick user manuals, but more often they are not documented at all, and have been discovered at a hard price as bugs of previously developed tests. They are perfect examples of the kind of precious domain-specific knowledge that expert designers accumulate over the years, and which is therefore particularly worthwhile to include in the test design environment for successive automatic reuse.

In the presented environment such properties are gathered as SLTL formulas (see below) in Constraint Libraries (see Fig. 4, right), which can be easily updated and which are automatically accessed by our model checker during the verification.

Definition 2 (SLTL).

The syntax of Semantic Linear-time Temporal Logic (SLTL) is given in BNF format by:

$$\Phi ::= \; A \mid \neg\Phi \mid (\Phi \wedge \Phi) \mid <c> \Phi \mid \mathbf{G}(\Phi) \mid (\Phi \mathbf{U} \Phi)$$

where A represents the set of atomic propositions over$S$, and c a possible branching condition expressed as a propositional logic formula over Act.

[2] The neighbourhood consists of the set of all the predecessors/successors of a test block along all paths in the model.

The SLTL formulas are interpreted over the set of all *legal test sequences*, i.e. alternating sequences of test blocks and conditions which start and end with test blocks. The semantics of SLTL formulas is now intuitively defined as follows:[3]

- A is satisfied by every test sequence whose first element (a test block) satisfies the atomic proposition A. Atomic propositions capture local test block specifications formulated as simple propositional logic formulas.
- Negation $\neg$ and conjunction $\wedge$ are interpreted in the usual fashion.
- Next-time operator $< >$:
  $<c> \Phi$ is satisfied by all test sequences whose second element (the first condition) satisfies c and whose *continuation*[4] satisfies Φ. In particular, $<tt> \Phi$ is satisfied by every test sequence whose continuation satisfies Φ.
- Generally operator $\mathbf{G}$:
  $\mathbf{G}(\Phi)$ requires that Φ is satisfied for every suffix.
- Until operator $\mathbf{U}$:
  $(\Phi \mathbf{U} \Psi)$ expresses that the property Φ holds at all test blocks of the sequence, until a position is reached where the corresponding continuation satisfies the property Ψ. Note that $\Phi \mathbf{U} \Psi$ guarantees that the property Ψ holds eventually (strong until).

The definitions of continuation and suffix may seem complicated at first. However, thinking in terms of paths within a flow graph clarifies the situation: a subpath always starts with a node (a test block) again (see also Fig. 3).

The interpretation of the logic over test models is defined path-wise: a test model satisfies a SLTL formula if all its paths do.

The introduction of *derived operators* supports a modular and intuitive formulation of complex properties. Convenient are the dual operators:

Disjunction: $\Phi \vee \Psi =_{df} \neg(\neg\Phi \wedge \neg\Psi)$
Eventually: $\mathbf{F}(\Phi) =_{df} \neg\mathbf{G}(\neg\Phi) = (tt\ \mathbf{U}\ \Phi)$

5 Typical Constraints

This section summarizes some typical constraints in order to illustrate the style and common patterns in temporal constraint specification for test cases. Technically, the following examples comprise modalities and examples for constraints written in our first-order extension of SLTL [3]. From the application point of view, we distinguish classes of constraints concerning different aspects of the application domain, which in particular differ in their scoping, i.e. not necessarily every test case must fulfill all constraints because it depends on the test purpose, which constraints are bound to the test case, see Sect. 6.

[3] A formal definition of the semantics, complete with taxonomies, can be found in [17].
[4] This continuation is simply the test sequence starting from the second test block.

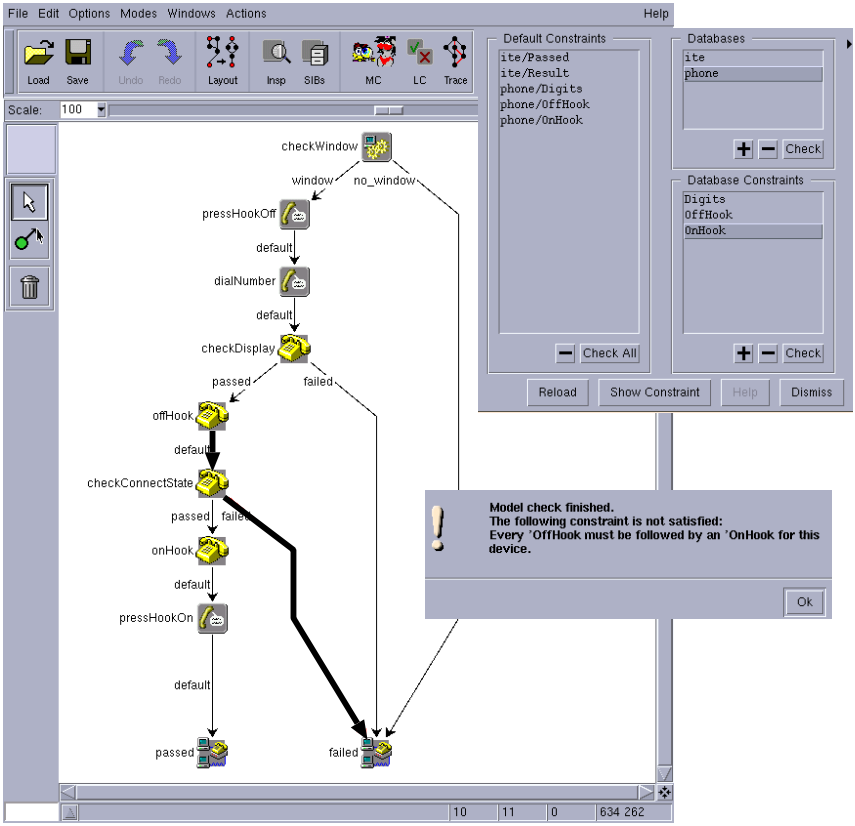


Fig. 4. Model Checking a Test Case

5.1 Legal Test Cases

This constraint class defines the characteristics of a correct test case, independently of any particular *SUT* and test purpose. Specifically, testing implies an evaluation of the runs wrt. expected observations done through *verdicts*. Precondition for automated testing is the presence of evaluation points along *each* path in the test graph. Additionally, to enable an automated evaluation of results, verdict points should be disposed in a *nonambiguous* and *noncontradictory way* along each path. To be expressive enough, a test graph should also foresee both possible verdicts. This is captured as follows:

– 'Every test run, i.e. every path in the test graph, must at least encounter passed or failed.' This constraint expresses the property that every possible path in the test graph rates the execution.

$$\mathsf{start} \;\Rightarrow\; \mathbf{F}\;(\mathsf{passed} \;\vee\; \mathsf{failed})$$

– 'Every test graph must contain at least one path which encounters passed.' This constraint captures the intuition that every test graph must provide the possibility to exit successfully, i.e., with verdict passed. Since we here consider linear time logic, in this paper we must express this property indirectly by disproving[5]

$$\mathsf{start} \Rightarrow \mathbf{G}\ (\neg\ \mathsf{passed})$$

– 'Once a verdict is assigned it cannot be changed.' This constraint ensures that verdict points should be disposed in a *nonambiguous* and *noncontradictory way* along each path, where

$$\mathsf{passed} \vee \mathsf{failed}\ \Rightarrow <tt>\ \mathbf{G}\ (\neg\mathsf{passed}\ \vee\ \neg\mathsf{failed})$$

Usually, such constraints are not explicitly formulated anywhere, since they are mostly obvious for test engineers. However, being able to formulate them in an automatically verifiable way changes this situation, because it is no longer a matter of mere understanding but of a drastically reduced search effort.

5.2 POTS Test Cases

This constraint class defines the characteristics of correct functioning of Plain Old Telephone Services (POTS), which build the basis of any CTI application behaviour. This constraint class is still very general, and in practice relevant for each specific test scenario we consider. In the following, we consider some constraints explaining the end-user level behaviour of telephones:

– 'Digits test blocks are only allowed to appear after a corresponding offHook test block', making sure that the phone is always properly initialized.

$$\forall n.\ \mathsf{start} \Rightarrow (\ \neg\mathsf{digits}(n)\ \mathbf{U}\ \mathsf{offHook}(n)\ )$$

– 'Every offHook must be followed by an onHook for this device'.

$$\forall n.\ \mathsf{offHook}(n)\ \Rightarrow\ \mathbf{F}\ (\ \mathsf{onHook}(n)\ )$$

– 'An onHook can only be executed when an offHook was initiated for this device before'.

$$\forall n.\ \mathsf{start} \Rightarrow\ (\ \neg\mathsf{onHook}(n)\ \mathbf{U}\ \mathsf{offHook}(n)\ )$$

In fact Fig. 4 demonstrates a model checker-produced counterexample for the second property in terms of a violating path.

Other constraints of this class concern the different signalling and communication channels of a modern phone with an end user: signalling via tones,

[5] Our model checker actually also covers the full mu-calculus and could therefore address a direct formulation of this property.

messaging via display, optic signalling via LEDs, vibration alarm. They must all convey correct and consistent information, and possible degradation of service in exceptional cases must respect consistency and follow a set of well-defined precedence rules. For example, tones have highest priority, and in case of further urgent signalling a display message may be suppressed but not overtaken.

Many other, somewhat more technical, constraints arise as soon as we consider also the switch side of the system. They concern, e.g.:

- details of the communication protocols which regulates the communication between the switch and peripherals (here simple telephones),
- the specific kind (analog, digital) of the protocol (e.g. ISDN, X25,...)
- the layer in the protocol stack
- variations of the protocol implementation specific to the vendor (here, Siemens), maybe even specific to single products or product lines.

One sees immediately that manageable organization of the constraints is a key characteristic of a practicable solution.

5.3 Service Specific Test Cases

Modern midrange telephone systems for private use or in small businesses already include an amazing number of additional services in addition to POTS. For example: signalling of an additional incoming call (a feature known as *Call Waiting*), for ISDN systems the display of the caller's number for incoming calls (*Calling Number Delivery*), conference calls (*Three Way Calling*), forwarding to other single or groups of numbers (*Call Forwarding*), embedded answering machine functionality (*Voice Mail*), and many more.

Testing such a telephone system means evaluating the correct behaviour of a set of phones in the context of one or more of those activated features, which can be again described as a collection of sets of constraints [4,5].

5.4 *SUT* Specific Test Cases

Considering concrete CTI settings like the one described in Fig. 2, we additionally need constraints about the correct initialization and functioning

- of the single units of the *SUT* (e.g. single CTI applications, or the switch),
- of the corresponding test tools,
- and of their interplay.

To give an idea of the complexity of the testing scenarios considered in practice, the concrete application already investigated in depth includes settings with a switch, up to eight phones of different nature, a complex CTI application with a server PC and a client PC serving several end-users, a running application suite, consisting of five programs, where the phones and the PC clients interact, and two test tools. The scenario is described in more detail in [9].

6 Test Suite Development in the Integrated Testing Environment

The resulting overall lifecycle for test development using the ABC-based *Test Coordinator* is two-dimensional: both the application and the environment can grow and be enriched during the development process.

6.1 Test Case and Test Suite Development

Based on libraries of test blocks and constraints, an initial test graph is graphically constructed via drag-and-drop from the test block palette and subsequently validated either via graph tracing or under model checking control. In discussions with test engineers and test designers it turned out that only very few simple patterns of constraints are required in order to express most of the desired properties. Thus, end users of the testing environment should be able to input their own constraints on the basis of very few corresponding templates without requiting the help of experts in temporal logic. Test suites are then composed of suitable sets of test cases to cover a certain application.

In the environment, distinct Constraint Libraries concern different *aspects* of the application: Figure 4, on the right, shows that the generic test library (`ite`) and the POTS libraries (`phone`) are currently loaded, and the specific constraints from those libraries have been selected for checking.

Typically, a test graph is built and modified by test designers in an *aspect-driven* fashion: the testing expert chooses one or more constraints of interest, expressing single *aspects* of the test case under construction, checks online the correctness of the current test graph and modifies it in case of mistakes. This cycle is iterated until all relevant aspects have been treated. Due to the online verification with the model checker, constraint violations are immediately detected.

6.2 Strengthening the Testing Environment

The test graph and test suite development is superposed by an orthogonal process of incremental strengthening of the application-specific environment: this happens by successively adding further test blocks and consistency constraints. Both strengthenings proceed naturally, on demand, along the evolution of needs.

Strengthening the Model. Entire new palettes of test blocks may turn out to become necessary when the environment or the application grow, e.g. to accommodate new test tools or new *SUT*'s (e.g., peripherals to the switch, mobile phones, applications, ...). Sometimes new single test blocks may also be needed e.g. when new releases of subsystems modify or extend their functionality.

New test blocks are also defined for the purpose of better test organization (when it becomes clear that certain test graph fragments have a high potential for reuse). The latter situation is supported by ABC's macro facility [19], which

essentially allows test designers to encapsulate (fragments of) test graphs as single, reusable test blocks. In the further development process, these blocks can be used just as 'ordinary' test blocks as shown in Fig. 3 (left): the test block marked with 'M' contains the full subgraph for Led and Display checks.

Strengthening the Constraints. New constraints naturally arise when considering single new *aspects* of the current systems, and new Constraint Libraries when considering new applications or ways of interactions. Here we use SLTL constraints to describe in an abstract and loose way single *aspects* of the behaviour of a complex, distributed, heterogeneous system, which we can access only in a blackbox or at best graybox fashion. This looseness is essential as most of the hardware is sealed, and most of the software is third party.

7 Conclusions

We have presented a new coarse grain approach to automated integrated testing, which combines library-based test design, incremental formalization, and library-based consistency checking via model checking. The impact of these features for the overall test process has been illustrated along an industrial application: an automated integrated testing environment for CTI-Systems. In fact, the gentle entry into practice due to incremental formalization was a 'conditio sine qua non' for the acceptance of the environment, because there has been some negative experience with formal methods in the past. The main reason for the failure of the prior approaches was their need of a full (formal) description of the *SUT*, which does not exists. Together with the early prototyping ability of the ABC, which allowed us to present a small running application within a couple of weeks, this won overall confidence for a long-term cooperation. Now, nine months after the first meeting, a prototype installation is already used by test engineers in Siemens' test laboratories.

We are convinced that our system will significantly reduce the testing effort, as already the untrained use of the only partially instantiated *integrated test environment* (ITE) led to noticeable reduction of the originally manual testing time. We are currently enforcing both training of the testers on the ITE, and refined instantiation with more test blocks, version information, and configuration constraints. This does not require any changes on the ITE itself, and we are optimistic that we will be able to give evidence in the very near future that these improvements indeed dramatically strengthen the impact of the ITE. Generalizations of the currently considered functional tests to other forms of testing like performance tests and stress tests are also possible, but require more effort and are planned for a successive project phase.

References

1. V. Braun, T. Margaria, B. Steffen, H. Yoo: *Automatic Error Location for IN Service Definition*, Proc. AIN'97, 2^{nd} Int. Workshop on Advanced Intelligent Networks, Cesena, 1997.

2. J.-C. Fernandez, C. Jard, T. Jéron and C. Viho: *An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology*, Science of Computer Programming, 29, 1997.
3. J. Hofmann: *Program Dependent Abstract Interpretation*, Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, August 1997.
4. B. Jonsson, T. Margaria, G. Naeser, J. Nyström, B. Steffen: *On Modelling Feature Interactions in Telecommunications*, Proc. of Nordic Workshop on Programming Theory, eds. B. Victor and W. Yi, 1999.
5. B. Jonsson, T. Margaria, G. Naeser, J. Nyström, B. Steffen: *Incremental Requirements Specification of Evolving Systems*, Feature Interactions in Telecommunications and Software Systems VI, eds. M. Calder and E. Magill, ISO Press, 2000.
6. T. Margaria, B. Steffen: *Backtracking-free Design Planning by Automatic Synthesis in METAFrame* Proc. FASE'98, Int. Conf. on Fundamental Aspects of Software Engineering, Lisbon, Apr. 1998, LNCS 1382, pp.188-204, Springer Verlag.
7. Mercury Interactive: Winrunner. `http://www.winrunner.com`
8. O. Niese, T. Margaria, M. Nagelmann, B. Steffen, G. Brune, H.-D. Ide: *An open Environment for Automated Integrated Testing*, 4^{th} Int. Conf. on Software and Internet Quality Week Europe (QWE'00), Brussels (Belgium), November 2000, CD-ROM Proccedings, pp 584-593.
9. O. Niese, M. Nagelmann, A. Hagerer, K. Strunck, W. Goerigk, A. Erochok, B. Hammelmann: *Demonstration of an Automated Integrated Testing Environment for CTI Systems*, Proc. FASE 2001, this volume. Genova (I), 2001.
10. Object Management Group: *The Common Object Request Broker: Architecture and Specification*, Revision 2.3, Object Management Group, 1999.
11. The Raise Project homepage. `http://dream.dai.ed.ac.uk/raise/`
12. Rational: The Rational Suite description. `http://www.rational.com/products`.
13. M. Schmitt, B. Koch, J. Grabowski, and D. Hogrefe, *Autolink - A Tool for Automatic and Semi-automatic Test Generation from SDL-Specifications*, Technical Report A-98-05, Medical Univ. of Lübeck, Germany, 1998.
14. ITU-T Recommendation Z.100, *CCITT specification and description language*, '93.
15. Sun: *Java Remote Method Invocation*. `http://java.sun.com/products/jdk/rmi`.
16. B. Steffen, T. Margaria, A. Claßen, V. Braun: *Incremental Formalization: a Key to Industrial Success*, in "Software: Concepts and Tools", Vol.17(2), pp. 78-91, Springer Verlag, July 1996.
17. B. Steffen, T. Margaria, V. Braun: *The Electronic Tool Integration Platform: Concepts and Design* Int. Journ. on Software Tools for Technology Transfer (STTT), Vol. 1 N. 1+2, Springer Verlag, November 1997, pp. 9-30.
18. B. Steffen, T. Margaria: *METAFrame in Practice: Intelligent Network Service Design*, In *Correct System Design – Issues, Methods and Perspectives*, LNCS 1710, Springer Verlag, 1999, pp.390-415.
19. B. Steffen, T. Margaria, V. Braun, and N. Kalt: *Hierarchical service definition*, *Annual Review of Communication, Int. Engineering Consortium (IEC), Chicago (USA)*, pages 847–856, 1997.
20. C. Stirling: *Modal and Temporal Logics*, In *Handbook of Logics in Computer Science*, Vol. 2, pp. 478 – 551, Oxford Univ. Press, 1995.
21. Telelogic: *Telelogic Tau*. `http://www.telelogic.com`.
22. J. Tretmans and A. Belinfante: *Automatic testing with formal methods*, In *EuroSTAR'99:* 7^{th} *European Int. Conference on Software Testing, Analysis & Review.* EuroStar Conferences, Galway, Ireland, November 8–12, 1999.

Demonstration of an Automated Integrated Testing Environment for CTI Systems

Oliver Niese[1], Markus Nagelmann[1], Andreas Hagerer[1],
Klaus Kolodziejczyk-Strunck[2], Werner Goerigk[3],
Andrei Erochok[3], and Bernhard Hammelmann[3]

[1] METAFrame Technologies GmbH, Dortmund, Germany
{ONiese, MNagelmann, AHagerer}@METAFrame.de
[2] HeraKom GmbH, Essen, Germany
klausk@herakom.de
[3] Siemens AG, Witten, Germany
{Werner.Goerigk, Andrei.Erochok, Bernhard.Hammelmann}@wit.siemens.de

1 An Integrated Testing Environment Based on ABC

We demonstrate the *Integrated Testing Environment*, ITE, an environment for automated and integrated testing at system level. A companion paper [4] describes the problem of systems integrated testing in its conceptual application modelling focus. Here and in [5], the concept's implementation is exemplified in case of automated integrated testing of a Computer Telephony Integration (CTI) system. The CTI system consists of a switch, an automatic call distributor, and a suite of applications forming a call center and supporting the tasks of a center's human agent, e.g., applications that enable an agent to log-on/log-off at the call distributor, or to initiate conference calls with other agents.

In the ITE, each hardware and software component of the CTI system is controlled by its own test tool, e.g., a proprietary hardware tracer for the switch or a GUI test tool such as *Rational SQA Robot* [6] for the applications. Coordinating which action has to be performed by which test tool is under responsibility of the *Test Coordinator*, a tool built on top of METAFrame Technologies' *Agent Building Center* (ABC) [2], a general-purpose environment for specification and verification of complex workflows.

As outlined in the following sections, the ITE supports the design of test cases including the verification of their consistency, the interactive combination of test cases resulting in test scenarios, and the execution of test cases and test scenarios in the heterogenous processing environment of CTI systems.

2 Design Support Features

System testing is characterized by focussing on inter-components cooperation. For the design of appropriate system-level test cases it is necessary to know what features the system provides, how to operate the system in order to stimulate a feature, and how to determine if features work. This information is gathered

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 249–252, 2001.

and after identification of the system's controllable and observable interfaces it is transformed into a set of stimuli (inputs) and verification actions (inspection of outputs, investigation of components' states). For each action a test block is prepared: a name and a class characterizing the block are specified and a set of formal parameters is defined to enable a more general usage of the block. In this way, for the CTI system to be tested a library of test blocks has been issued that includes test blocks representing and implementing, e.g.

Common actions. Initialization of test tools, system components, test cases and general reporting functions,

Switch-specific actions. Initialization of switches of different extensions,

Call-related actions. Initiation and pick up of calls via a PBX-network or a local switch,

CTI application-related actions. Miscellaneous actions to operate a CTI application via its graphical user interface, e.g., log-on/log-off of an agent, establish a conference party, initiate a call via a GUI, or check labels of GUI-elements.

The library of test blocks grows dynamically whenever new actions are made available.
The design of test cases consists in the behaviour-oriented combination of test blocks. This combination is done graphically, i.e., icons representing test blocks are graphically stuck together to yield test graph structures that embody the test behaviour in terms of control.

3 Verification Support Features

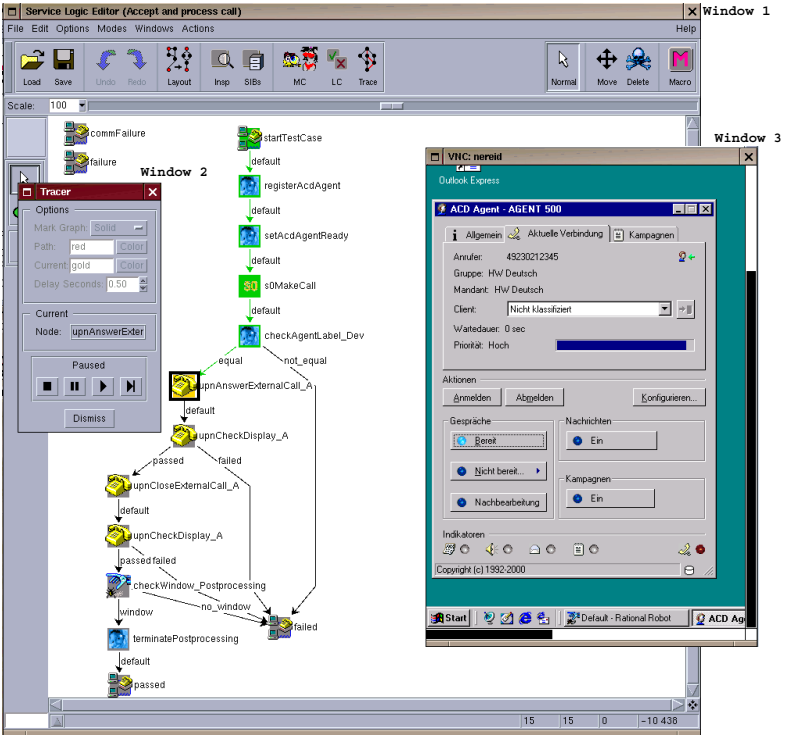
As explained thoroughly in [4], design of test cases is constantly accompanied by online verification of the global correctness and consistency of the test cases' control flow logic. During the design phase, vital properties concerning the usage of parameters (local properties) and concerning the interplay between the stimuli and verification actions of a test case (global properties) can be verified. Design decisions that conflict with the constraints and consistency conditions of the intended system are thus immediately detected.

Local properties specified imperatively are used to check that the parameters are correctly set. Global properties concerning the interactions between arbitrarily distant test blocks of a test graph are expressed in a user-friendly specification language based on the Semantic Linear-time Temporal Logic [3], and are gathered in a constraint library accessed by the environment's model checker during verification. Basic constraints have been formulated for the CTI system, e.g., if a test graph includes an action "hook-on", this must be preceded by an action "hook-off".

If the model checker detects an inconsistency, a plain text explanation of the violated constraint appears. In addition, test blocks violating a local property as well as paths violating a global property are marked.

4 Execution Support Features

In the ITE, test cases can be executed immediately by means of ABC's tracer. Starting at a dedicated test block of a test graph the tracer proceeds from test block to test block. The actions represented by a test block are performed, i.e., stimuli and inspection requests are sent to the corresponding system's component, responses are received, evaluated, and the evaluation result is used to select one of the possibilities to pass control flow to a succeeding test block.



The screen shot illustrates this. The system-under-test is a call center application, in this case a client-server CTI application called "ACD Agent" which runs on different computers than the *Test Coordinator*. In this test we emulate a human call center agent with identifier *AGENT 500* and handle some actions via the agent's GUI of the PC application. The test case graph is shown in Window 1. The Tracer-window (Window 2) controls the execution of the test case. The test block executed last is highlighted in the test graph window. Window 3 shows the agent's computer screen.

The implementation of this execution scheme requires two activities during set-up of the ITE. First, the actions referenced via test blocks have to be implemented by means of test tools. This task is performed by test engineers which are

familiar with test tools, their handling and programming. For each action, the test engineers has to specify instructions to be executed by the test tool determined to support the specific action, e.g., via recording GUI-activities. Second, specific tracer code that is assigned to the action's test block and that will be executed by the tracer has to be developed. Experience with the CTI system shows that this code can be generated automatically for most actions. Manual development is necessary only if the test block shall initiate the execution of multiple actions in order to meet real-time requirements or if more complex evaluation of information about a component's state or reaction is required.

To communicate with different test tools, a flexible CORBA-based interface has been designed for the ITE. This interface comprises basic methods which all test tools have to support as well as special features of tools added by means of specializations. In the *Test Coordinator*, these methods are wrapped in a uniform way and are provided for implementation of tracer code in form of adapters. The extensibility of the ITE by additional adapters for other test tools is the key of the approach. On the test tool's side, the interface functionality has to be provided either by implementing a plug-in including a CORBA object request broker or by implementing a separate server process that communicates with the test tool via a dedicated interface (e.g., COM/DCOM).

At the moment, two test tools have been made accessible in the ITE: *Rational SQA Robot* [6] for driving an application's GUI and a proprietary tool *Hipermon* developed by HeraKom GmbH. This tool controls a device simulator that communicates with the switch and that is able to accurately emulate calls.

Finally, when executing a test graph, a detailed protocol is prepared. For each test block the tracer executes, all relevant data (its execution time, its name, the version of the files associated with the test block, the block's parameter values, and the processed data) are written to the protocol.

References

1. S. Gladstone: *Testing Computer Telephony Systems and Networks*, Telecom Books, 1996.
2. B. Steffen, T. Margaria, V. Braun, N. Kalt: *Hierarchical Service Definition*, Annual Review of Communic., Vol. 51, Int. Engineering Consortium, Chicago, 1997, pp.847-856.
3. T. Margaria, B. Steffen: *Backtracking-free Design Planning by Automatic Synthesis in METAFrame* Proc. FASE'98, Int. Conf. on Fundamental Aspects of Software Engineering, Lisbon, LNCS 1382, Springer Verlag, 1998, pp.188-204.
4. O. Niese, B. Steffen, T. Margaria, A. Hagerer, G. Brune, H.-D. Ide: *Library-based Design and Consistency Checking of System-level Industrial Test Cases*, FASE 2001, this volume.
5. O. Niese, T. Margaria, A. Hagerer, M. Nagelmann, B. Steffen, G. Brune, H.-D. Ide: *An Automated Testing Environment for CTI Systems Using Concepts for Specification and Verification of Workflows*, accepted for publication in Annual Review of Communic., Vol. 54, Int. Engineering Consortium, Chicago, 2000.
6. Rational, Inc.: *The Rational Suite description*, `http://www.rational.com/products`.

Semantics of Architectural Specifications in CASL

Lutz Schröder[1], Till Mossakowski[1], Andrzej Tarlecki[2,3], Bartek Klin[4], and Piotr Hoffman[2]

[1] BISS, Department of Computer Science, Bremen University
[2] Institute of Informatics, Warsaw University
[3] Institute of Computer Science, Polish Academy of Sciences
[4] BRICS, Arhus University

Abstract. We present a semantics for architectural specifications in CASL, including an extended static analysis compatible with model-theoretic requirements. The main obstacle here is the lack of amalgamation for CASL models. To circumvent this problem, we extend the CASL logic by introducing enriched signatures, where subsort embeddings form a category rather than just a preorder. The extended model functor has amalgamation, which makes it possible to express the amalgamability conditions in the semantic rules in static terms. Using these concepts, we develop the semantics at various levels in an institution-independent fashion.

1 Introduction

A common feature of present-day algebraic specification languages (see e.g. [Wir86,EM85,GHG+93,CoFI96,SW99]) is the provision of operations for building large specifications in a structured fashion from smaller and simpler ones [BG77]. For the quite different purpose of describing the modular structure of software systems under development [SST92], architectural specifications have been introduced as a comparatively novel feature in the algebraic specification language CASL recently developed by the CoFI group [CoF,CoF99a,Mos99].

The main idea is that architectural specifications describe branching points in system development by indicating units (modules) to be independently developed and showing how these units, once developed, are to be put together to produce the overall result. Semantically, units are viewed as given models of specifications, to be used as building blocks for models of more complex specifications, e.g. by amalgamating units or by applying parametrized units. Architectural specifications have been introduced and motivated in [BST99].

The aim of the present paper is to outline the semantics of architectural specifications at various levels. We use a simple subset of CASL architectural specifications, which is expressive enough to study the main mechanisms and features of the semantics. A crucial prerequisite for a semantics of architectural specifications is the amalgamation property, which allows smaller models to be combined into larger ones under statically checkable conditions. Somewhat informally, the amalgamation property ensures that whenever two models 'share'

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 253–268, 2001.

components in the intersection of their signatures, they can be unambiguously put together to form a model of the union of their signatures. Many standard logical systems (like multisorted equational [EM85] and first-order logic [MT93] with the respective standard notions of model) admit amalgamation, so quite often this property is taken for granted in work on specification formalisms (cf. e.g. [ST88]). However, the expected amalgamation property fails to hold for the logical system underlying CASL.

We develop a three-step semantics that circumvents this problem. The first step is a purely model-theoretic semantics. Here, amalgamability is just *required* whenever it is needed. This makes the definition of the semantics as straightforward and permissive as possible, but leaves the task of actually checking these model-theoretic requirements. Thus, the natural second step is to give a semantics of architectural specifications in terms of diagrams which express the sharing that is present in the unit declarations and definitions. This allows us to reformulate the model-theoretic amalgamability conditions in 'almost' static terms. A suitable amalgamation property is needed to make the static character of these conditions explicit. The trick used in the third step to achieve this is to embed the CASL logic into a richer logic that does have amalgamation. This makes it possible to restate the amalgamability conditions as entirely static factorization properties of signature morphisms.

These three steps of the semantics are in fact independent of the details of the underlying CASL logical system: we present them in the framework of an arbitrary logical system formalized as an institution [GB92]. As a result, the factorization properties to which we reduce the amalgamation conditions are still relatively abstract. A calculus for checking these factorization properties in the case of the specificic logic underlying CASL is developed in a separate paper [KHT$^+$].

We refer to [Mac97,AHS90] for categorical terminology left unexplained here.

2 Architectural Specifications

As indicated above, architectural specifications in CASL provide a means of stating how implementation units are used as building blocks for larger components. (Dynamic interaction between modules and dynamic changes of software structure are currently beyond the scope of this approach.)

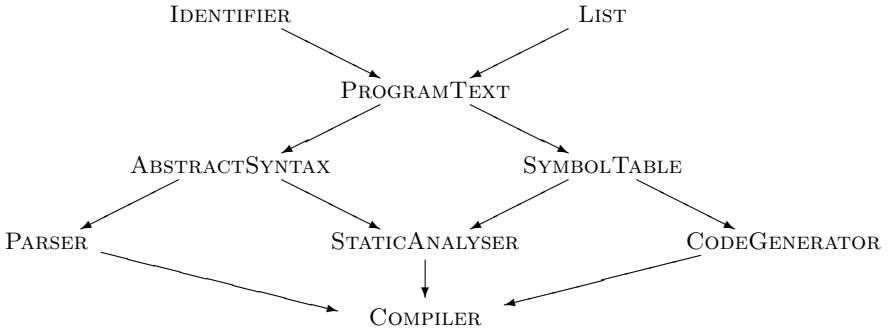
Units are represented as names to which a specification is assigned. Such a named unit is to be thought of as a given model of the specification. Units may be parametrized, where specifications are assigned to both the parameters and the result. The result specification is required to extend the parameter specifications. A parametrized unit is to be understood as a function which, given models of the parameter specifications, outputs a model of the result specification; this function is required to be *persistent*, i.e. reducing the result to the parameter signatures reproduces the parameters.

Units can be assembled via unit expressions which may contain operations such as renaming or hiding of symbols, amalgamation of units, and application

of a parametrized unit. Terms containing such operations will only be defined if symbols that are identified, e.g. by renaming them to the same symbol or by amalgamating units that have symbols in common, are also interpreted in the same way in all 'collective' models of the units defined so far.

An architectural specification consists in declaring or defining a number of units, as well as in providing a way of assembling them to yield a result unit.

Example 1. A (fictitious) specification structure for a compiler might look roughly as follows:



(The arrows indicate the extension relation between specifications.) An architectural specification of the compiler in CASL [CoF99a] might have the following form:

arch spec BUILDCOMPILER =
units I : IDENTIFIER **with sorts** *Identifier*, *Keyword*;
L : ELEM $\to$ LIST[ELEM];
PT = $L[I$ **fit sort** *Elem* $\mapsto$ *Identifier*$]$
and $L[I$ **fit sort** *Elem* $\mapsto$ *Keyword*$]$;
AS : ABSTRACTSYNTAX **given** PT;
ST : SYMBOLTABLE **given** PT;
P : PARSER **given** AS;
SA : STATICANALYSER **given** AS, ST;
CG : CODEGENERATOR **given** ST
result P **and** SA **and** CG
end

(Here, the keyword **with** is used to just list some of the defined symbols. The keyword **given** indicates imports.) According to the above specification, the parser, the static analyser, and the code generator would be constructed building upon a given abstract syntax and a given mechanism for symbol tables, and the compiler would be obtained by just putting together the former three units. Roughly speaking, this is only possible (in a manner that can be statically checked) if all symbols that are shared between the parser, the static analyser and the code generator already appear in the units for the abstract syntax or the symbol tables.

In order to keep the presentation as simple as possible, we consider a modified sublanguage of CASL architectural specifications:

Architectural specifications: $ASP ::= \mathbf{arch\ spec}\ UDD^*\ \mathbf{result}\ T$;
$UDD ::= Dcl \mid Dfn$
An architectural specification consists of a list of unit declarations and definitions followed by a unit result term.

Unit declarations: $Dcl ::= U\colon SP \mid U\colon SP_1 \xrightarrow{\tau} SP_2$
A unit declaration introduces a unit name with its type, which is either a specification or a specification of a parametrized unit, determined by a specification of its parameter and its result, which extends the parameter via a signature morphism τ — we assume that the definition of specifications and some syntactic means to present signature morphisms are given elsewhere. (By resorting to explicit signature morphisms, we avoid having to discuss the details of signature inclusions.)

Unit definitions: $Dfn ::= U = T$
A unit definition introduces a (non-parametrized) unit and gives its value by a unit term.

Unit terms: $T ::= U \mid U[T\ \mathbf{fit}\ \sigma] \mid T_1\ \mathbf{with}\ \sigma_1\ \mathbf{and}\ T_2\ \mathbf{with}\ \sigma_2$
A unit term is either a (non-parametrized) unit name, or a unit application with an argument that fits via a signature morphism σ, or an amalgamation of units via signature morphisms σ_1 and σ_2. We require that σ_1 and σ_2 form an episink (have a common target signature and are jointly epi); we thus slightly generalize the amalgamation operation of CASL here, again avoiding the need to present the details of signature unions (cf. [Mos00]).

Imports as used in Example 1 can be regarded as syntactical sugar for a parametrized unit which is instantiated only once.

3 Institutions and Amalgamation

The semantic considerations ahead rely on the notion of *institution* [GB92]. An institution I consists of a category **Sign** of *signatures*, a *model functor*

$$\mathbf{Mod} : \mathbf{Sign}^{op} \to \mathbf{CAT},$$

where **CAT** denotes the quasicategory of categories and functors [AHS90], and further components which formalize sentences and satisfaction. In this context, we need only the model functor. For a signature Σ, $\mathbf{Mod}(\Sigma)$ is referred to as the category of *models for* Σ, and for a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$, $\mathbf{Mod}(\sigma) : \mathbf{Mod}(\Sigma_2) \to \mathbf{Mod}(\Sigma_1)$ is called the *reduct functor*. $\mathbf{Mod}(\sigma)(M)$ is often written as $M|_\sigma$.

A cocone for a diagram in **Sign** is called *amalgamable* if it is mapped to a limit under **Mod**. I has the *(finite) amalgamation property* if (finite) colimit cocones are amalgamable, i.e. if **Mod** preserves (finite) limits.

The underlying logic of CASL is formalized by the institution $SubPCFOL$ (for 'subsorted partial first order logic with sort generation constraints'); the associated signature category is denoted by **CASLsign** [CoF99b,Mos]. As mentioned above, $SubPCFOL$ does not have the finite amalgamation property (cf. Example 3).

4 Basic Architectural Semantics

We now proceed to give a *basic semantics* of the architectural language defined in Section 2 similarly as for full CASL [CoF99b]. We use the natural semantics style, by presenting rules for the *static semantics*, with judgements written as $\_ \vdash \_ \rhd \_$, and for the *model semantics*, with judgements written as $\_ \vdash \_ \Rightarrow \_$ (where the blank spaces represent, in this order, a context of some kind, a syntactical object, and a semantical object). We simplify the rules of the model semantics by assuming a successful application of the corresponding rules of the static semantics, with symbols introduced there available for the model semantics as well. Moreover, we will regard $\mathbf{Mod}(\Sigma)$ as a *class* of models for the purposes of the model semantics.

The static semantics for an architectural specification yields a static context describing the signatures of the units declared or defined within the specification and the signature of its result unit. Thus, a *static context* $C_{st} = (P_{st}, B_{st})$ consists of two finite maps: P_{st} from unit names to parametrized unit signatures, which in turn are signature morphisms $\tau : \Sigma_1 \to \Sigma_2$, and B_{st} from unit names to signatures (for non-parametrized units). We require the domains of P_{st} and B_{st} to be disjoint. The empty static context that consists of two empty maps will be written as $C_{st}^{\emptyset}$. Given an initial static context, the static semantics for unit declarations and definitions produces a static context by adding the signature for the newly introduced unit, and the static semantics for unit terms determines the signature for the resulting unit.

In terms of the model semantics, a (non-parametrized) unit M over a signature Σ is just a model $M \in \mathbf{Mod}(\Sigma)$. A parametrized unit F over a parametrized unit signature $\tau : \Sigma_1 \to \Sigma_2$ is a persistent partial function $F : \mathbf{Mod}(\Sigma_1) \rightharpoonup \mathbf{Mod}(\Sigma_2)$ (i.e. $F(M)|_\tau = M$ for each $M \in dom\, F$); the domain of F is determined by the argument *specification*.

The model semantics for architectural specifications yields a *unit context* $\mathcal{C}$, which is a class of *unit environments* E, i.e. finite maps from unit names to units as introduced above, and a *unit evaluator* UEv, a function that yields a unit when given a unit environment in the unit context. The unconstrained unit context, which consists of all environments, will be written as $\mathcal{C}^{\emptyset}$. The model semantics for unit declarations and definitions enlarges unit contexts as expected. Finally, the model semantics for a unit term yields a unit evaluator, given a unit context.

The complete semantics is given in Figure 1, where we use some auxiliary notation: given a unit context $\mathcal{C}$, a unit name U and a class of units $\mathcal{V}$,

$$\mathcal{C} \times \{U \mapsto \mathcal{V}\} := \{E + \{U \mapsto V\} \mid E \in \mathcal{C}, V \in \mathcal{V}\},$$

where $E + \{U \mapsto V\}$ maps U to V and otherwise behaves like E. Moreover, given a unit context $\mathcal{C}$, a unit name U and a unit evaluator UEv,

$$\mathcal{C} \otimes \{U \mapsto UEv\} := \{E + \{U \mapsto UEv(E)\} \mid E \in \mathcal{C}\}.$$

We assume that the signature category is equipped with a partial *selection of pushouts* $(\sigma_R : \Sigma_1 \to \Sigma_R, \tau_R : \Sigma_2 \to \Sigma_R, \Sigma_R)$ for spans $(\sigma : \Sigma \to \Sigma_1, \tau : \Sigma \to$

Σ_2) of signature morphisms (where (σ, τ) may fail to have a selected pushout even when has a pushout). In CASL, the selected pushouts would be the ones that can be expressed by signature translations and simple syntactic unions. We also assume that the semantics for specifications is given elsewhere, with $\vdash SP \rhd \Sigma$ and $\vdash SP \Rightarrow \mathcal{M}$ implying $\mathcal{M} \subseteq \mathbf{Mod}(\Sigma)$.

Perhaps the only points in the semantics that require some discussion are the rules of the model semantics for unit application and amalgamation.

In the rule for application of a parametrized unit U, we have the requirement

$$\text{for each } E \in \mathcal{C},\ UEv(E)|_\sigma \in dom\, E(U),$$

where UEv denotes the unit evaluator and $\mathcal{C}$ the unit context. This is just the statement that the fitting morphism correctly 'fits' the actual parameter as an argument for the parametrized unit. To verify this requirement, one typically has to prove that σ is a specification morphism from the argument specification to the specification of the actual parameter (which, in the general case, has to be constructed from the relevant unit term by means of a suitable calculus). In general, this requires some semantic or proof-theoretic reasoning.

The situation is different with the conditions marked with a $(*)$ in Figure 1. These 'amalgamability conditions' are typically expected to be at least partially discharged by some static analysis — similarly to the sharing requirements present in some programming languages (cf. e.g. Standard ML [Pau96]). Of course, the basic static analysis given here is not suited for this purpose, since no information is stored about dependencies between units. This will be taken care of in the second level of the semantics.

5 Extended Static Architectural Semantics

As a solution to the problem just outlined, we now introduce an extended static analysis that keeps track of sharing among the units by means of a diagram of signatures; the idea here is that a symbol shares with any symbol to which it is mapped under some morphism in the diagram.

For our purposes, it suffices to regard a diagram as a graph morphism $D : \mathbf{I} \to \mathbf{Sign}$, where $\mathbf{I}$ is a directed graph called the *scheme* of the diagram. We use categorical terminology for $\mathbf{I}$, i.e. we call its nodes 'objects', its edges 'morphisms' etc., and we write $\mathrm{Ob}\,\mathbf{I}$ for the set of objects.

We will use the usual notion of extension for diagrams. Two diagrams D_1, D_2 *disjointly extend* D if both D_1 and D_2 extend D and moreover, the intersection of their schemes is the scheme of D. If this is the case then the union $D_1 \cup D_2$ is well-defined.

The judgements of the *extended static semantics* are written as $\_ \vdash \_ \Rrightarrow \_$. Most of the rules differ only formally from the rules for the static semantics; the essential differences are in the rules for unit terms. The extended static semantics additionally carries around the said diagram of signatures. Signatures for unit terms are associated to distinguished objects in the diagram scheme.

$$\frac{\vdash UDD^* \rhd C_{st} \qquad C_{st} \vdash T \rhd \Sigma}{\vdash \textbf{arch spec } UDD^* \textbf{ result } T \rhd (C_{st}, \Sigma)} \qquad \frac{\vdash UDD^* \Rightarrow \mathcal{C} \qquad \mathcal{C} \vdash T \Rightarrow UEv}{\vdash \textbf{arch spec } UDD^* \textbf{ result } T \Rightarrow (\mathcal{C}, UEv)}$$

$$\frac{\begin{array}{c} C_{st}^{\emptyset} \vdash UDD_1 \rhd (C_{st})_1 \\ \cdots \\ (C_{st})_{n-1} \vdash UDD_n \rhd (C_{st})_n \end{array}}{\vdash UDD_1 \ldots UDD_n \rhd (C_{st})_n} \qquad \frac{\begin{array}{c} \mathcal{C}^{\emptyset} \vdash UDD_1 \Rightarrow \mathcal{C}_1 \\ \cdots \\ \mathcal{C}_{n-1} \vdash UDD_n \Rightarrow \mathcal{C}_n \end{array}}{\vdash UDD_1 \ldots UDD_n \Rightarrow \mathcal{C}_n}$$

$$\frac{\vdash SP \rhd \Sigma \qquad U \notin (dom\ P_{st} \cup dom\ B_{st})}{(P_{st}, B_{st}) \vdash U\colon SP \rhd (P_{st}, B_{st} + \{U \mapsto \Sigma\})} \qquad \frac{\vdash SP \Rightarrow \mathcal{M}}{\mathcal{C} \vdash U\colon SP \Rightarrow \mathcal{C} \times \{U \mapsto \mathcal{M}\}}$$

$$\frac{\vdash SP_1 \rhd \Sigma_1 \qquad \vdash SP_2 \rhd \Sigma_2 \qquad \tau : \Sigma_1 \to \Sigma_2 \qquad U \notin (dom\ P_{st} \cup dom\ B_{st})}{(P_{st}, B_{st}) \vdash U\colon SP_1 \xrightarrow{\tau} SP_2 \rhd (P_{st} + \{U \mapsto \tau\}, B_{st})}$$

$$\frac{\begin{array}{c} \vdash SP_1 \Rightarrow \mathcal{M}_1 \qquad \vdash SP_2 \Rightarrow \mathcal{M}_2 \\ \mathcal{F} = \{F : \mathcal{M}_1 \to \mathcal{M}_2 \mid \text{for } M \in \mathcal{M}_1, F(M)|_\tau = M\} \end{array}}{\mathcal{C} \vdash U\colon SP_1 \xrightarrow{\tau} SP_2 \Rightarrow \mathcal{C} \times \{U \mapsto \mathcal{F}\}}$$

$$\frac{(P_{st}, B_{st}) \vdash T \rhd \Sigma \qquad U \notin (dom\ P_{st} \cup dom\ B_{st})}{(P_{st}, B_{st}) \vdash U = T \rhd (P_{st}, B_{st} + \{U \mapsto \Sigma\})} \qquad \frac{\mathcal{C} \vdash T \Rightarrow UEv}{\mathcal{C} \vdash U = T \Rightarrow \mathcal{C} \otimes \{U \mapsto UEv\}}$$

$$\frac{U \in dom\ B_{st}}{(P_{st}, B_{st}) \vdash U \rhd B_{st}(U)} \qquad \frac{}{\mathcal{C} \vdash U \Rightarrow \lambda E \in \mathcal{C} \cdot E(U)}$$

$$\frac{\begin{array}{c} P_{st}(U) = \tau : \Sigma_1 \to \Sigma_2 \qquad C_{st} \vdash T \rhd \Sigma^A \qquad \sigma : \Sigma_1 \to \Sigma^A \\ (\sigma_R, \tau_R, \Sigma_R) \text{ is the selected pushout of } (\sigma, \tau) \end{array}}{(P_{st}, B_{st}) \vdash U[T \textbf{ fit } \sigma] \rhd \Sigma_R}$$

$$\frac{\begin{array}{c} \mathcal{C} \vdash T \Rightarrow UEv \\ \text{for each } E \in \mathcal{C},\ UEv(E)|_\sigma \in dom\ E(U) \\ \left.\begin{array}{c} \text{for each } E \in \mathcal{C},\ \text{there is a unique } M \in \mathbf{Mod}(\Sigma_R) \text{ such that} \\ M|_{\tau_R} = UEv(E) \text{ and } M|_{\sigma_R} = E(U)(UEv(E)|_\sigma) \end{array}\right\} (*) \\ UEv_R = \{E \mapsto M \mid E \in \mathcal{C}, M|_{\tau_R} = UEv(E), M|_{\sigma_R} = E(U)(UEv(E)|_\sigma)\} \end{array}}{\mathcal{C} \vdash U[T \textbf{ fit } \sigma] \Rightarrow UEv_R}$$

$$\frac{\begin{array}{c} C_{st} \vdash T_1 \rhd \Sigma_1 \qquad C_{st} \vdash T_2 \rhd \Sigma_2 \\ \sigma_1 : \Sigma_1 \to \Sigma \text{ and } \sigma_2 : \Sigma_2 \to \Sigma \text{ form an episink} \end{array}}{(P_{st}, B_{st}) \vdash T_1 \textbf{ with } \sigma_1 \textbf{ and } T_2 \textbf{ with } \sigma_2 \rhd \Sigma}$$

$$\frac{\begin{array}{c} \mathcal{C} \vdash T_1 \Rightarrow UEv_1 \qquad \mathcal{C} \vdash T_2 \Rightarrow UEv_2 \\ \left.\begin{array}{c} \text{for each } E \in \mathcal{C},\ \text{there is a unique } M \in \mathbf{Mod}(\Sigma) \text{ such that} \\ M|_{\sigma_i} = UEv_i(E),\ i = 1, 2 \end{array}\right\} (*) \\ UEv = \{E \mapsto M \mid E \in \mathcal{C} \text{ and } M|_{\sigma_i} = UEv_i(E),\ i = 1, 2\} \end{array}}{\mathcal{C} \vdash T_1 \textbf{ with } \sigma_1 \textbf{ and } T_2 \textbf{ with } \sigma_2 \Rightarrow UEv}$$

Fig. 1. Basic semantics

Explicitly, an *extended static context* $\mathcal{C}_{st} = (P_{st}, \mathcal{B}_{st}, D)$ consists of a map P_{st} that assigns unit signatures to parametrized unit names (as before), a signature diagram D, and a map $\mathcal{B}_{st}$ that assigns objects of the diagram scheme to (non-parametrized) unit names. As before, we require that the domains of P_{st} and $\mathcal{B}_{st}$ are disjoint. $\mathcal{C}_{st}$ determines a static context $ctx(\mathcal{C}_{st})$ formed by extracting the signature information for non-parametrized unit names from the diagram and forgetting the diagram itself. The empty extended static context, which consists of two empty maps and the empty diagram, is written as $\mathcal{C}_{st}^{\emptyset}$. The extended static semantics for unit declarations and definitions expands the given extended static context; for unit terms, it extends the signature diagram and indicates an object in the scheme that represents the result.

The diagrams enable us to restate the amalgamability conditions in a static way: for any diagram $D : \mathbf{I} \to \mathbf{Sign}$, $\langle M_i \rangle_{i \in \mathrm{Ob}\,\mathbf{I}}$ is called *consistent* with D if for each $i \in \mathrm{Ob}\,\mathbf{I}$, each $M_i \in \mathbf{Mod}(D(i))$, and for each $m : i \to j$ in $\mathbf{I}$, $M_i = M_j|_{D(m)}$. We denote the class of all model families that are consistent with D by $\mathbf{Mod}(D)$. Then D *ensures amalgamability for* D', where D' extends D, if any family in $\mathbf{Mod}(D)$ can be uniquely extended to a family in $\mathbf{Mod}(D')$.

Although we have formulated this property in terms of model families, it is essentially static: the class of model families considered is not restricted by axioms, but only by morphisms between signatures. The static nature of this condition will be made explicit in Section 7.

The rules of the extended static semantics are listed in Figure 2; given the heuristics provided above, they should be largely self-explanatory. However, the relationship between the basic static and model semantics and the extended static semantics requires a few comments.

Since, as stated at the end of the previous section, the correctness condition for arguments of parametrized units cannot be disposed of statically, one cannot expect that the extended static semantics is stronger than the model semantics, i.e. that its successful application guarantees that the model sematics will succeed as well. However, this is almost true in the sense that argument fitting is the only point that is left entirely to the model semantics. Formally, this can be captured by the statement that, assuming a successful run of the extended static semantics, the conditions marked with a $(*)$ in the rules of the model semantics (cf. Figure 1) can be removed.

Calling the combination of the extended static semantics and the thus simplified model semantics *extended semantics*, we now indeed have:

Theorem 2. *If the extended semantics of an architectural specification is defined, then the basic semantics is defined as well and yields the same result.*

Of course, no completeness can be expected here: even if the basic semantics is successful for a given phrase, the extended semantics may fail. This happens if the model-theoretic amalgamability conditions hold due to axioms in specifications rather than due to static properties of the involved constructions.

An additional source of failures of the extended static semantics is that we have deliberately chosen a so-called generative static analysis: the results of applications of parametrized units 'share' with other units in the signature diagram

$$\frac{\vdash UDD^* \mathrel{|\!\!\gg} \mathcal{C}_{st} \qquad \mathcal{C}_{st} \vdash T \mathrel{|\!\!\gg} (i, D)}{\vdash \textbf{arch spec}\ UDD^*\ \textbf{result}\ T \mathrel{|\!\!\gg} (ctx(\mathcal{C}_{st}), D(i))}$$

$$\frac{\begin{array}{c}\mathcal{C}^{\emptyset}_{st} \vdash UDD_1 \mathrel{|\!\!\gg} (\mathcal{C}_{st})_1 \\ \cdots \\ (\mathcal{C}_{st})_{n-1} \vdash UDD_n \mathrel{|\!\!\gg} (\mathcal{C}_{st})_n\end{array}}{\vdash UDD_1 \ldots UDD_n \mathrel{|\!\!\gg} (\mathcal{C}_{st})_n}$$

$$\frac{\begin{array}{c}\vdash SP \rhd \Sigma \qquad U \not\in (dom\, P_{st} \cup dom\, \mathcal{B}_{st}) \\ D' \text{ results from } D \text{ by adding a new object } i \text{ with } D'(i) = \Sigma\end{array}}{(P_{st}, \mathcal{B}_{st}, D) \vdash U\colon SP \mathrel{|\!\!\gg} (P_{st}, \mathcal{B}_{st} + \{U \mapsto i\}, D')}$$

$$\frac{\begin{array}{c}\vdash SP_1 \rhd \Sigma_1 \qquad \vdash SP_2 \rhd \Sigma_2 \qquad \tau : \Sigma_1 \to \Sigma_2 \\ U \not\in (dom\, P_{st} \cup dom\, \mathcal{B}_{st})\end{array}}{(P_{st}, \mathcal{B}_{st}, D) \vdash U\colon SP_1 \xrightarrow{\tau} SP_2 \mathrel{|\!\!\gg} (P_{st} + \{U \mapsto \tau\}, \mathcal{B}_{st}, D)}$$

$$\frac{(P_{st}, \mathcal{B}_{st}, D) \vdash T \mathrel{|\!\!\gg} (i, D') \qquad U \not\in (dom\, P_{st} \cup dom\, \mathcal{B}_{st})}{(P_{st}, \mathcal{B}_{st}, D) \vdash U = T \mathrel{|\!\!\gg} (P_{st}, B_{st} + \{U \mapsto i\}, D')}$$

$$\frac{U \in dom\, B_{st}}{(P_{st}, \mathcal{B}_{st}, D) \vdash U \mathrel{|\!\!\gg} (\mathcal{B}_{st}(U), D)}$$

$$\frac{\begin{array}{c}P_{st}(U) = \tau : \Sigma_1 \to \Sigma_2 \qquad C_{st} \vdash T \mathrel{|\!\!\gg} (i, D) \qquad \sigma : \Sigma_1 \to D(i) \\ (\sigma_R, \tau_R, \Sigma_R) \text{ is the selected pushout of } (\sigma, \tau) \\ D' \text{ results from } D \text{ by adding new objects } j, k \\ \text{and new morphisms } m : j \to i, n : j \to k \text{ with } D'(m) = \sigma, D'(n) = \tau \\ D'' \text{ results from } D' \text{ by adding a new object } l \\ \text{and new morphisms } r : i \to l, s : k \to l \text{ with } D''(r) = \tau_R, D''(s) = \sigma_R \\ D' \text{ ensures amalgamability for } D''\end{array}}{(P_{st}, \mathcal{B}_{st}, D) \vdash U[T\ \textbf{fit}\ \sigma] \mathrel{|\!\!\gg} (l, D'')}$$

$$\frac{\begin{array}{c}(P_{st}, \mathcal{B}_{st}, D) \vdash T_1 \mathrel{|\!\!\gg} (i_1, D_1) \qquad (P_{st}, \mathcal{B}_{st}, D) \vdash T_2 \mathrel{|\!\!\gg} (i_2, D_2) \\ \sigma_1 : D_1(i_1) \to \Sigma \text{ and } \sigma_2 : D_2(i_2) \to \Sigma \text{ form an episink} \\ D_1 \text{ and } D_2 \text{ are disjoint extensions of } D \\ D' \text{ results from } D_1 \cup D_2 \text{ by adding a new object } j \\ \text{and new morphisms } m_1 : i_1 \to j, m_2 : i_2 \to j \text{ with } D'(m_1) = \sigma_1,\ D'(m_2) = \sigma_2 \\ D_1 \cup D_2 \text{ ensures amalgamability for } D'\end{array}}{(P_{st}, \mathcal{B}_{st}, D) \vdash T_1\ \textbf{with}\ \sigma_1\ \textbf{and}\ T_2\ \textbf{with}\ \sigma_2 \mathrel{|\!\!\gg} (j, D')}$$

Fig. 2. Extended static semantics

constructed only via the morphisms from the parameter signatures to the actual arguments. Thus, two applications of the same unit to the same argument need not 'share'. As a consequence, the amalgamability condition of the extended static semantics may fail for them, while the corresponding condition in the basic model semantics would clearly hold. A 'non-generative' (or 'applicative') version of the extended static semantics is sketched in Remark 11 below.

The motivation for this choice is the fact that many typical programming languages we aim at (notably, Standard ML [Pau96]) impose such a 'generative' semantics in their static analysis — working with more permissive conditions here would make our architectural specifications incompatible with the modularization facilities of such languages.

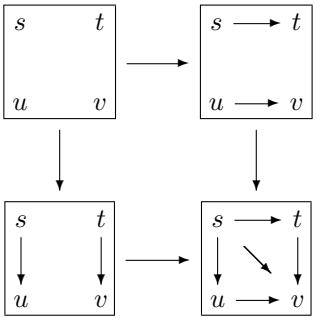
However, we will see in Section 6 (Th. 6) that — generativity issues aside — we have as much completeness as one may hope for, i.e. that the extended static semantics detects all the amalgamation that can be established statically.

6 Enriched Signatures

The actual verification of the amalgamability conditions in the extended static semantics is precisely the point where the fact that amalgamation fails in the CASL institution begins to cause difficulties:

Example 3. Assume that the specification LIST[ELEM] that appears in Example 1 provides a type *List*[*Elem*] of lists of type *Elem*. Recall that the specification of identifiers introduces two sorts *Identifier* and *Keyword*, and that the parametrized unit L ('list') is applied to these two sorts in the 'program text' unit PT.

Now suppose that the specifier of PARSER decides that key words should be treated as identifiers, so that *Keyword* $<$ *Identifier* and *List*[*Keyword*] $<$ *List*[*Identifier*] . Suppose, moreover, that the specifier of STATICANALYSER finds it convenient to code simple elements as lists in some way, i.e. *Identifier* $<$ *List*[*Identifier*] and *Keyword* $<$ *List*[*Keyword*]. Singling out the union P **and** SA from the term defining the compiler in Example 1, we thus obtain a diagram of CASL signatures for the union that has the following (abstracted) form, where the arrows within the squares represent subsort embeddings:

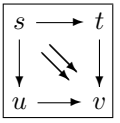


(1)

Even though the above diagram is in fact a pushout in the category **CASLsign**, compatible models of the component signatures cannot in general be amalgamated, since the composed subsort embeddings $s < t < v$ and $s < u < v$ in the result need not be the same. (Consequently, the extended static semantics defined in the previous section fails here.)

This observation suggests that one should enlarge the signature category in such a way that the above square is no longer a pushout, and that, moreover,

the enlarged category would have to take the fact into account that there may, in general, be more than one embedding between two sorts. Thus, the 'correct' pushout signature would have the form



. (2)

Motivated by this example, we introduce a category **enrCASLsign** of *enriched signatures* in which **CASLsign** can be represented via a functor

$$\Phi : \mathbf{CASLsign} \to \mathbf{enrCASLsign}.$$

Moreover, we equip this category with a model functor

$$\mathbf{Mod}_e : \mathbf{enrCASLsign}^{op} \to \mathbf{CAT}$$

which has the amalgamation property (i.e. preserves limits) and which 'extends' the model functor **Mod** of the CASL institution $SubPCFOL$, i.e. $\mathbf{Mod}_e \circ \Phi^{op}$ and **Mod** are naturally isomorphic. One can build an institution around **enrCASLsign** and define an institution representation of $SubPCFOL$ therein in the sense of [Tar96]. At any rate, we shall use terms like 'amalgamable' w.r.t. $\mathbf{Mod}_e$. The details of the definitions and full proofs will be presented separately [SMT$^+$]; the basic concepts are outlined below.

As suggested by the example, the step from **CASLsign** to **enrCASLsign** chiefly consists in replacing the preorder on the sorts by a sort category (category sorted algebras, although without a view on amalgamation, go back to [Rey80]). The fact that all embeddings are actually interpreted as injective maps is reflected by the requirement that all morphisms in the sort category are monomorphisms. There is an elegant way to handle overloading of function and predicate symbols in this setting using left and right actions of the sort category on the symbols; see [SMT$^+$] for details.

Now sets and partial maps form a (rather large) enriched signature $\mathbf{Set_p}$: take injective maps as the sort category, relations as predicate symbols, and total (partial) functions as total (partial) function symbols.

Signature morphisms are defined in the obvious way, i.e. as consisting of a functor between the sort categories and maps between the respective symbol classes which are compatible with symbol profiles and 'overloading' (i.e. with the mentioned actions of the sort category). These data define the signature category **enrCASLsign**, where objects have to be restricted to *small* signatures (i.e. signatures where the sort category and the symbol classes are small) in order to actually obtain a category. It is easily seen that **enrCASLsign** is cocomplete, and that, just as with small and large categories, colimits in **enrCASLsign** remain colimits in the world of 'large enriched signatures' such as $\mathbf{Set_p}$.

Models of an enriched signature Σ can now be defined as signature morphisms

$$\Sigma \to \mathbf{Set_p}.$$

Model morphisms are defined by the standard homomorphy conditions. Signature morphisms induce reduct functors in the opposite direction via composition; this defines the model functor $\mathbf{Mod}_e$.

The functor $\Phi : \mathbf{CASLsign} \to \mathbf{enrCASLsign}$ acts on CASL signatures by interpreting the sort preorder as a thin category (and by completing the symbol sets as required by the mentioned actions; cf. [SMT$^+$]).

Now it is easily verified that one indeed has a natural isomorphism $\mathbf{Mod}_e \circ \Phi^{op} \to \mathbf{Mod}$. Thus:

Proposition 4. *A cocone in* $\mathbf{CASLsign}$ *is amalgamable in SubPCFOL iff its image under* Φ *is amalgamable w.r.t.* $\mathbf{Mod}_e$.

Thanks to the way models are defined, amalgamation for enriched signatures comes almost for free: models are given by a representable functor (namely, $hom(\_, \mathbf{Set_p})$), which automatically preserves limits; little more consideration has to be given to model morphisms. Explicitly:

Proposition 5. $\mathbf{Mod}_e$ *has the amalgamation property.*

Moreover, the 'converse' of amalgamation holds as well:

Theorem 6. $\mathbf{Mod}_e$ *reflects isomorphisms.*

Since $\mathbf{Mod}_e$ preserves limits and $\mathbf{enrCASLsign}^{op}$ is complete, it follows that $\mathbf{Mod}_e$ reflects limits. Explicitly, a cocone in $\mathbf{enrCASLsign}$ is amalgamable iff it is a colimit. This is essentially what is meant by the 'optimal degree of completeness' statement in Section 5: a cocone in $\mathbf{CASLsign}$ is amalgamable iff it is mapped to a colimit under Φ.

7 Static Analysis via Enriched Signatures

We are now ready to translate the amalgamation conditions that appear in the rules for unit application and amalgamation in the extended static semantics to entirely static conditions. To this end, we assume that we have a cocomplete category $\mathbf{EnrSign}$ of enriched signatures with a model functor $\mathbf{Mod}_e : \mathbf{EnrSign}^{op} \to \mathbf{CAT}$ which has the amalgamation property and reflects isomorphisms (limits) and a functor $\Phi : \mathbf{Sign} \to \mathbf{EnrSign}$ such that $\mathbf{Mod}_e \circ \Phi^{op}$ and $\mathbf{Mod} : \mathbf{Sign}^{op} \to \mathbf{CAT}$ are naturally isomorphic. For the CASL institution *SubPCFOL*, these data have been constructed above.

Recall that the said amalgamation conditions required diagrams to ensure amalgamability for certain extensions, which was defined as unique extendability of consistent families of models. By the assumption on the model functors, this requirement is equivalent to the corresponding statement for the translations of the diagrams via Φ. By the amalgamation property, a consistent family of models for a diagram D in $\mathbf{EnrSign}$ is essentially the same as a model of the colimit signature $\operatorname{colim} D$. Thus we have

Proposition 7. *Let D' be a diagram in* **EnrSign** *that extends D. D ensures amalgamation for D' iff the induced morphism* $\operatorname{colim} D \to \operatorname{colim} D'$ *is an isomorphism.*

This condition can be checked by means of a factorization property in the cases of interest here:

Definition 8. Let $\mathbf{A}$ be a category, and let $D' : \mathbf{I}' \to \mathbf{A}$ be a diagram that extends $D : \mathbf{I} \to \mathbf{A}$. Then D *covers* D' if, for each $j \in \operatorname{Ob} \mathbf{I}'$, the sink of all $D'(m) : D(i) \to D'(j)$, where $i \in \operatorname{Ob} \mathbf{I}$ and $m : i \to j$ in $\mathbf{I}'$, is an episink.

Proposition 9. *Let D and D' be diagrams in a cocomplete category, where D' extends D. If D covers D', then the induced morphism* $\operatorname{colim} D \to \operatorname{colim} D'$ *is an isomorphism iff the colimit cocone for D extends to a cocone for D'.*

In the two cases where amalgamation is required in the rules of the extended static semantics, the covering condition is satisfied. Thus we have essentially reduced the amalgamation problem to proving the existence of the factorizations required in the above proposition. In both cases, the factorization condition concerns a sink (τ_1, τ_2) in **Sign** (in the case of amalgamation, the two injections into the union, and in the case of application, the pushout cocone):

$$\begin{array}{ccccc} \Phi(D(i_1)) & \xrightarrow{\Phi(\tau_1)} & \Phi(\Sigma) & \xleftarrow{\Phi(\tau_2)} & \Phi(D(i_2)) \\ & \searrow^{\mu_{i_1}} & \vdots\,\theta & \swarrow_{\mu_{i_2}} & \\ & & \operatorname{colim} \Phi \circ D & & \end{array}$$

(D denotes the original diagram, and the μ_i denote the colimit injections).

Example 10. The simple union of sort preorders presented in Example 3, Diagram (1), fails to admit a factorization as above, since the colimit will have two different sort embeddings $s \to v$ as depicted in Diagram (2).

In order to provide a construction for the factorization θ in the concrete case of **enrCASLsign**, we have to require additionally that $(\Phi(\tau_1), \Phi(\tau_2))$ is an extremal episink, i.e. that the images of $\Phi(\tau_1)$ and $\Phi(\tau_2)$ jointly generate $\Phi(\Sigma)$. This is the case for pushouts and unions in **CASLsign** (although unions need not be extremal in **CASLsign**!).

Under this additional condition, it is clear how θ has to be defined if it exists, namely by extending the effect of the μ_i to terms formed from the generators. The task that remains is to check if this results in a well-defined signature morphism. This requires a calculus for proving equality of morphisms and symbols in the colimit; such a calculus is discussed in [KHT$^+$].

Remark 11. In the construction of a non-generative semantics for unit application (cf. Section 5), the amalgamation property provides an easy criterion for the equivalence of two instantiations: let U be a parametrized unit over the signature $\tau : \Sigma_1 \to \Sigma_2$. Two actual argument models are considered to be (partially)

equivalent if they reduce (via fitting morphisms $\sigma_i : \Sigma_1 \to D(j_i)$, $i = 1, 2$, where D denotes the present context diagram) to the same model of the parameter signature Σ_1. This will be the case for all pairs of models that appear in consistent families for D if

$$\mu_{j_1} \circ \Phi(\sigma_1) = \mu_{j_2} \circ \Phi(\sigma_2),$$

where μ is the colimit cocone for $\Phi \circ D$. In this case, we can use the same edge of the diagram scheme to represent τ in both applications of U; this has the effect that the two results with signatures Σ_R^1 and Σ_R^2 share to exactly the right degree via the maps $\Sigma_2 \to \Sigma_R^i$, $i = 1, 2$, that appear in the defining pushouts.

8 Conclusions and Future Work

We have presented and discussed the semantics of a small and modified but quite representative subset of CASL architectural specifications in an institution-independent way. Besides the basic static and model semantics, we have laid out an extended static analysis, where sharing information between models is stored as a diagram of signatures. This has allowed us to formulate the required amalgamability conditions 'almost' statically, i.e. without referring to particular models constructed. In institutions with amalgamation, these conditions can be replaced by literally static ones; this may require representing the given institution in one that has the amalgamation property.

We have demonstrated how such a representation can be defined for the standard CASL institution; the main point here was that one has to admit categories of subsort embeddings instead of just preorders in the signatures. Computational aspects of the amalgamability conditions for this specific institution are discussed separately [KHT+]. It is known that these conditions are in general undecidable. However, decision procedures can be developed for important special cases; for instance, orders (rather than preorders) of sorts admit deciding the amalgamability conditions by a polynomial algorithm [Kli00]. For the general case, it seems that an approximative algorithm that restricts the length of 'cells' involved in the proofs will suffice in practically relevant cases.

The technique for the extended static analysis with colimits of enriched signatures providing a way to statically reformulate amalgamability conditions is general enough to be readily available for other design specification frameworks where amalgamation causes problems. In fact, we conjecture that any institution can be canonically extended to an institution with the amalgamation property by adding formal colimits of signatures.

The work presented here is independent of the logical structure of institutions — sentences and satisfaction do not play any explicit role here (except for being used implicitly in basic specifications, of course). However, the sentences become relevant as soon as we discuss further issues of verification in architectural specifications (represented here by the remaining fitting condition in the semantics of unit applications). As proposed in [Hof00], formal proof obligations can be extracted from such conditions using colimits of specification diagrams,

but only if the underlying institution has the amalgamation property. The technique proposed here should allow us to circumvent this requirement: specification diagrams can be translated to the enriched signature category and put together there, opening a way also for the development of tools supporting validation and verification of CASL (architectural) specifications.

Acknowledgements. Partial support by the CoFI Working Group (ESPRIT WG 29432) is gratefully acknowledged, as well as the work of the CoFI Language and Semantics Task Groups. Moreover, we wish to thank Michel Bidoit and Donald Sannella for collaboration on architectural specifications and Maura Cerioli for pointing out the lack of amalgamation for CASL models.

References

[CoFI96] CoFI. Catalogue of existing frameworks. Catalogue.html, in [CoF], 1996.

[AHS90] J. Adámek, H. Herrlich, and G. E. Strecker. *Abstract and Concrete Categories*. Wiley Interscience, New York, 2nd edition, 1990.

[BG77] R. M. Burstall and J. A. Goguen. Putting theories together to make specifications. In *Proc. 5th Intl. Joint Conf. on Artificial Intelligence*, pages 1045–1058. Carnegie-Mellon University, 1977.

[BST99] M. Bidoit, D. Sannella, and A. Tarlecki. Architectural specifications in CASL. In *7th Intl. Conference on Algebraic Methodology and Software Technology (AMAST'98)*, volume 1548 of *LNCS*, pages 341–357. Springer, 1999.

[CoF] CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible by WWW[1] and FTP[2].

[CoF99a] CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language – Summary, version 1.0. Documents/CASL/Summary, in [CoF], July 1999.

[CoF99b] CoFI Semantics Task Group. CASL – The CoFI Algebraic Specification Language – Semantics. Note S-9 (version 0.96), in [CoF], July 1999.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics*. Springer, 1985.

[GB92] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39:95–146, 1992.

[GHG+93] J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer, 1993.

[Hof00] P. Hoffman. Semantics of architectural specifications (in Polish). Master's thesis, Warsaw University, 2000.

[KHT+] B. Klin, P. Hoffman, A. Tarlecki, T. Mossakowski, and L. Schröder. Checking amalgamability conditions for CASL architectural specifications. Work in progress; see also [Kli00].

[1] http://www.brics.dk/Projects/CoFI

[2] ftp://ftp.brics.dk/Projects/CoFI

[Kli00] B. Klin. An implementation of static semantics for architectural specifications in CASL (in Polish). Master's thesis, Warsaw University, 2000.

[Mac97] S. Mac Lane. *Categories for the Working Mathematician*. Springer, Berlin, 2nd edition, 1997.

[Mos] T. Mossakowski. Relating CASL with other specification languages: the institution level. *Theoretical Computer Science*. To appear.

[Mos99] P. D. Mosses. CASL: A guided tour of its design. In *13th Workshop on Abstract Datatypes WADT 98*, volume 1589 of *LNCS*, pages 216–240. Springer, 1999.

[Mos00] T. Mossakowski. Specification in an arbitrary institution with symbols. In *14th Workshop on Abstract Datatypes WADT 99*, volume 1827 of *LNCS*, pages 252–270. Springer, 2000.

[MT93] K. Meinke and J. V. Tucker, editors. *Many-sorted Logic and its Applications*. Wiley, 1993.

[Pau96] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, New York, 2nd edition, 1996.

[Rey80] J. C. Reynolds. Using category theory to design implicit conversions and generic operators. In *Semantics-Directed Compiler Generation*, volume 94 of *LNCS*, pages 211–258. Springer, 1980.

[SMT$^+$] L. Schröder, T. Mossakowski, A. Tarlecki, P. Hoffman, and B. Klin. Amalgamation via enriched signatures. Work in progress.

[SST92] D. Sannella, S. Sokołowski, and A. Tarlecki. Towards formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Informatica*, 29:689–736, 1992.

[ST88] D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76:165–210, 1988.

[SW99] D. Sannella and M. Wirsing. Specification languages. In A. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specifications*, pages 243–272. Springer, 1999.

[Tar96] A. Tarlecki. Moving between logical systems. In *11th Workshop on Specification of Abstract Data Types*, volume 1130 of *LNCS*, pages 478–502. Springer, 1996.

[Wir86] M. Wirsing. Structured algebraic specifications: A kernel language. *Theoretical Computer Science*, 42:123–249, 1986.

Extending Development Graphs with Hiding

Till Mossakowski[1], Serge Autexier[2], and Dieter Hutter[3]

[1] BISS, Dept. of Computer Science, University of Bremen, P.O. Box 330 440, D 28334 Bremen, till@tzi.de, Fax: +49 421 218 3054
[2] FR 6.2 Informatik, Saarland University, P.O. Box 15 11 50, D 66041 Saarbrücken, Autexier@ags.uni-sb.de, Fax: +49 681 302 2235
[3] DKFI GmbH, Stuhlsatzenhausweg 3, D 66123 Saarbrücken, hutter@dfki.de, Fax: +49 681 302 2235

Abstract. Development graphs are a tool for dealing with structured specifications in a formal program development in order to ease the management of change and reusing proofs. In this work, we extend development graphs with hiding (e.g. hidden operations). Hiding is a particularly difficult to realize operation, since it does not admit such a good decomposition of the involved specifications as other structuring operations do. We develop both a semantics and proof rules for development graphs with hiding. The rules are proven to be sound, and also complete relative to an oracle for conservative extensions. We also show that an absolute complete set of rules cannot exist. The whole framework is developed in a way independent of the underlying logical system (and thus also does not prescribe the nature of the parts of a specification that may be hidden).

1 Introduction

It has long been recognized that *specifications in the large* are only manageable if they are built in a structured way. Specification languages, like CASL [CASL98], provide various mechanisms to combine basic specifications to structured specifications. Analogously, verification tools have to provide appropriate mechanisms to structure the corresponding logical axiomatizations. In practice, a formal program development is an evolutionary process [VSE96]. Specification and verification are mutually intertwined. Failed proofs give rise to changes of the specification which in turn will render previously found proofs invalid. For practical purposes it is indispensable to restrict the effects of such changes to a minimum in order to preserve as much proof effort as possible after a change of the specification.

Various structuring operations have been proposed (e.g. [DGS91,ST88,ST92]) in order to modularize specifications and proof systems have been described to deal with them (e.g. [CM97,HWB97]). Traditionally, the main motivations for modularization have been the sharing of sub-specifications within one specification, the reuse of specifications, and the structuring of proof obligations as well as applicable lemmas. However, the structure of specifications can also be exploited when the effects of changes are analyzed.

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 269–283, 2001.
Springer-Verlag Berlin Heidelberg 2001

In [AHMS00], development graphs have been introduced as a tool for dealing with structured specifications in a way easing management of change and reusing proofs. Also, a translation of structured specifications in CASL, an international standard for algebraic specification languages, to development graphs has been set up. However, this translation does not cover the case of hiding yet. In this work, we extend development graphs in a way that allows also to deal with hiding. Compared with other structuring operations like union, renaming and parameterization, hiding is a particularly difficult to realize operation. This is because hiding does not admit such a good decomposition of the involved specifications as other structuring operations do.

2 Motivation

As a running example consider the following example of specifying and refining a sorting function *sorter*.

Given some specification of total orders and lists, an abstract specification of this sorting function may be denoted in CASL syntax as follows:

spec SORTING
[TOTALORDER]
=
{
LIST [**sort** *Elem*]
then
preds $is\_ordered : List[Elem]$;
$permutation : List[Elem] \times List[Elem]$;
forall $x, y : Elem$;
$L, L1, L2 : List[Elem]$
- $is\_ordered([])$
- $is\_ordered([x])$
- $is\_ordered(x :: (y :: L)) \Leftrightarrow x \leq y \wedge is\_ordered(y :: L)$
- $permutation(L1, L2) \Leftrightarrow (\forall x : Elem \bullet x \in L1 \Leftrightarrow x \in L2)$

then
op $sorter : List[Elem] \to List[Elem]$;
forall $L : List[Elem]$
- $is\_ordered(sorter(L))$
- $permutation(L, sorter(L))$

}
hide *is_ordered*, *permutation*
end

is_ordered and *permutation* are auxiliary predicates to specify *sorter*, and are hidden to the outside. A model of this specification is just an interpretation of the *sorter* function (together with a model of the imported specifications of total orders and lists) that can be extended to a model of the whole specification (including *is_ordered* and *permutation*).

During a development, we may refine SORTING into a design specification describing a particular sorting algorithm. For simplicity, we choose a sorting algorithm which recursively inserts the head element in the sorted tail list. In CASL we obtain the following specification:

spec INSERTSORT
  [TOTALORDER]
  =
{
    LIST [**sort** *Elem*]
  then
    ops $insert : Elem \times List[Elem] \to List[Elem];$
      $sorter : List[Elem] \to List[Elem];$

    forall $x, y : Elem;$
      $L : List[Elem]$
    - $insert(x, []) = [x]$
    - $insert(x, y :: L) =$
      $x :: insert(y, L)$ *when* $x \leq y$ *else* $y :: insert(x, L)$
    - $sorter([]) = []$
    - $sorter(x :: L) = insert(x, sorter(L))$

}
hide *insert*
end

Now the interesting question arises whether INSERTSORT is actually a refinement of SORTING; i.e. whether each INSERTSORT-model is also a SORTING-model.

3 Preliminaries

When studying development graphs with hiding, we want to focus on the structuring and want to abstract from the details of the underlying logical system. Therefore, we recall the abstract notion of logic from Meseguer [Mes89]. Logics consist of model theory and proof theory. Model theory is captured by the notion of *institution*, providing an abstract framework for talking about signatures, models, sentences and satisfaction. Proof theory is captured by the notion of *entailment system*, providing an abstract framework for talking about signatures, sentences and entailment.

Let $\mathcal{CAT}$ be the category of categories and functors.[1]

Definition 1. *An* institution *[GB92]* $\mathcal{I} = (\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \models)$ *consists of*

- *a category* $\mathbf{Sign}$ *of* signatures,
- *a functor* $\mathbf{Sen}: \mathbf{Sign} \longrightarrow \mathbf{Set}$ *giving the set of* sentences $\mathbf{Sen}(\Sigma)$ *over each signature* Σ*, and for each signature morphism* $\sigma: \Sigma \longrightarrow \Sigma'$*, the sentence translation map* $\mathbf{Sen}(\sigma): \mathbf{Sen}(\Sigma) \longrightarrow \mathbf{Sen}(\Sigma')$*, where often* $\mathbf{Sen}(\sigma)(\varphi)$ *is written as* $\sigma(\varphi)$*,*

[1] Strictly speaking, $\mathcal{CAT}$ is not a category but only a so-called quasicategory, which is a category that lives in a higher set-theoretic universe.

- *a functor* $\mathbf{Mod}\colon \mathbf{Sign}^{op} \longrightarrow \mathcal{CAT}$ *giving the category of* models *over a given signature, and for each signature morphism* $\sigma\colon \Sigma \longrightarrow \Sigma'$, *the* reduct functor $\mathbf{Mod}(\sigma)\colon \mathbf{Mod}(\Sigma') \longrightarrow \mathbf{Mod}(\Sigma)$, *where often* $\mathbf{Mod}(\sigma)(M')$ *is written as* $M'|_\sigma$,
- *a satisfaction relation* $\models_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma)$ *for each* $\Sigma \in \mathbf{Sign}$,

such that for each $\sigma\colon \Sigma \longrightarrow \Sigma'$ *in* $\mathbf{Sign}$ $M' \models_{\Sigma'} \sigma(\varphi) \Leftrightarrow M'|_\sigma \models_\Sigma \varphi$ *holds for each* $M' \in \mathbf{Mod}(\Sigma')$ *and* $\varphi \in \mathbf{Sen}(\Sigma)$ *(*satisfaction condition*).*

Definition 2. *An* entailment system $\mathcal{E} = (\mathbf{Sign}, \mathbf{Sen}, \vdash)$ *consists of a category* $\mathbf{Sign}$ *of* signatures*, a functor* $\mathbf{Sen}\colon \mathbf{Sign} \longrightarrow \mathbf{Set}$ *giving the set of* sentences *over a given signature, and for each* $\Sigma \in |\mathbf{Sign}|$, *an entailment relation* $\vdash_\Sigma \subseteq |\mathbf{Sen}(\Sigma)| \times \mathbf{Sen}(\Sigma)$ *such that the following properties are satisfied:*

1. reflexivity: *for any* $\varphi \in \mathbf{Sen}(\Sigma)$, $\{\varphi\} \vdash_\Sigma \varphi$,
2. monotonicity: *if* $\Gamma \vdash_\Sigma \varphi$ *and* $\Gamma' \supseteq \Gamma$ *then* $\Gamma' \vdash_\Sigma \varphi$,
3. transitivity: *if* $\Gamma \vdash_\Sigma \varphi_i$, *for* $i \in I$, *and* $\Gamma \cup \{\varphi_i \mid i \in I\} \vdash_\Sigma \psi$, *then* $\Gamma \vdash_\Sigma \psi$,
4. $\vdash$-translation: *if* $\Gamma \vdash_\Sigma \varphi$, *then for any* $\sigma\colon \Sigma \longrightarrow \Sigma'$ *in* $\mathbf{Sign}$, $\sigma[\Gamma] \vdash_{\Sigma'} \sigma(\varphi)$.

We write $\Gamma^{\vdash_\Sigma}$ for $\{\varphi \mid \Gamma \vdash_\Sigma \varphi\}$.

Definition 3. *A* logic *is a 5-tuple* $\mathcal{LOG} = (\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \vdash, \models)$ *such that:*

1. $(\mathbf{Sign}, \mathbf{Sen}, \vdash)$ *is an entailment system (denoted by* $ent(\mathcal{LOG})$*),*
2. $(\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \models)$ *is an institution (denoted by* $inst(\mathcal{LOG})$*), and*
3. *the following* soundness condition *is satisfied: for any* $\Sigma \in |\mathbf{Sign}|$, $\Gamma \subseteq \mathbf{Sen}(\Sigma)$ *and* $\varphi \in \mathbf{Sen}(\Sigma)$, $\Gamma \vdash_\Sigma \varphi$ *implies* $\Gamma \models_\Sigma \varphi$

A logic is complete *if, in addition,* $\Gamma \models_\Sigma \varphi$ *implies* $\Gamma \vdash_\Sigma \varphi$.

Throughout the rest of the paper, we will work with an arbitrary but fixed logic $\mathcal{LOG} = (\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \vdash, \models)$ such that $\mathbf{Sign}$ has finite colimits, and $\mathcal{LOG}$ admits weak amalgamation, i.e. $\mathbf{Mod}$ maps finite colimits to weak limits.

Given a diagram $D\colon I \longrightarrow \mathbf{Sign}$, let us call a family $(m_i)_{i \in |I|}$ *consistent with* I, if for each $i \in |I|$, $m_i \in \mathbf{Mod}(D(i))$, and for each $l\colon i \longrightarrow j \in I$, $m_j|_{D(l)} = m_i$.

The weak amalgamation property can now be reformulated as follows: $\mathcal{LOG}$ admits weak amalgamation iff for each diagram $D\colon I \longrightarrow \mathbf{Sign}$ and each family $(m_i)_{i \in |I|}$ *consistent with* I, there exists a model $m \in Colim\ D$ with $m|_{\mu_i} = m_i$, where $\mu_i\colon D(i) \longrightarrow Colim\ D$ are the colimit injections.

There are plenty of logics satisfying the above requirements, e.g. many-sorted equational logic, many-sorted first-order logic, various temporal and object-oriented logics etc. The logic underlying CASL, subsorted partial first-order logic with sort generation constraints, does not admit weak amalgamation. However, the CASL logic can be embedded into a logic with a richer signature category and a model functor admitting (weak) amalgamation [SMHKT01]. Hence, the results of this paper also are applicable for CASL, if colimits are taken in the richer signature category.

4 Development Graphs with Hiding

A development graph, as introduced in [AHMS00], represents the actual state of a formal program development. It is used to encode the structured specifications in various phases of the development. Roughly speaking, each node of the graph represents a theory like for instance List, Sorting or InsertSort in Casl. The links of the graph define how theories can make use of other theories.

Leaves in the graph correspond to basic specifications, which do not make use of other theories (e.g. Total_order). Inner nodes correspond to structured specifications which define theories using other theories (e.g. Sorting using Total_order). The corresponding links in the graph are called *definition links*. Arising proof obligations are attached as so-called *theorem links* to this graph. We here add a new type of definition links corresponding to *hiding*.

Definition 4. *A* development graph *is an acyclic, directed graph* $\mathcal{S} = \langle \mathcal{N}, \mathcal{L} \rangle$.

$\mathcal{N}$ is a set of nodes. Each node $N \in \mathcal{N}$ is a tuple (Σ^N, Φ^N) such that Σ^N is a signature and $\Phi^N \subseteq \mathbf{Sen}(\Sigma^N)$ is the set of **local axioms** *of N.*

$\mathcal{L}$ is a set of directed links, so-called **definition links***, between elements of $\mathcal{N}$. Each definition link from a node M to a node N is either*

- **global** *(denoted $M \xrightarrow{\sigma} N$), annotated with a signature morphism $\sigma : \Sigma^M \to \Sigma^N$, or*
- **local** *(denoted $M \xrightarrow{\sigma} N$), again annotated with a signature morphism $\sigma : \Sigma^M \to \Sigma^N$, or*
- **hiding** *(denoted $M \xrightarrow[h]{\sigma} N$), annotated with a signature morphism $\sigma : \Sigma^N \to \Sigma^M$* going against the direction of the link.

To simplify matters, we write $M \xrightarrow{\sigma} N \in \mathcal{S}$ instead of $M \xrightarrow{\sigma} N \in \mathcal{L}$ when $\mathcal{L}$ are the links of $\mathcal{S}$.

In Fig. 1 we present the development graph for the running example: The left part of the graph represents the structured specification Sorting, and the the right part the structured specification InsertSort.

The next definition captures the existence of a path of local and global definition links between two nodes. Notice that such a path must not contain any hiding links.

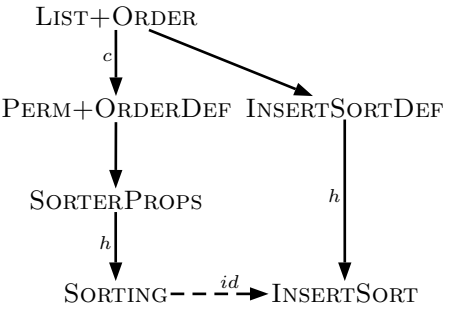


Fig. 1. Development graph for the sorting example

Definition 5. *Let $\mathcal{S}$ be a development graph. A node M is* **globally reachable** *from a node N via a mapping σ, $N \xrightarrow{\sigma} M \in \mathcal{S}$ for short, iff either $N = M$ and $\sigma = \lambda$, or $N \xrightarrow{\sigma'} K \in \mathcal{S}$, and $K \xrightarrow{\sigma''} M \in \mathcal{S}$, with $\sigma = \sigma'' \circ \sigma'$.*

A node M is **locally reachable** *from a node N via a mapping σ, $N \xrightarrow{\sigma} M \in \mathcal{S}$ for short, iff $N \xrightarrow{\sigma} M \in \mathcal{S}$ or there is a node K with $N \xrightarrow{\sigma'} K \in \mathcal{S}$, $K \xrightarrow{\sigma''} M \in \mathcal{S}$, such that $\sigma = \sigma'' \circ \sigma'$.*

Obviously global reachability implies local reachability since the theory of a node is defined with the help of local axioms.

Definition 6. *Let $\mathcal{S} = \langle \mathcal{N}, \mathcal{L} \rangle$ be a development graph. A node $N \in \mathcal{N}$ is* flatable *iff for all nodes $M \in \mathcal{N}$ with incoming hiding definition links, it holds that N is not reachable from M.*

The models of flatable nodes does not depend on existing hiding links. Thus we use the original approach of defining the theory of a node:

Definition 7. *Let $\mathcal{S} = \langle \mathcal{N}, \mathcal{L} \rangle$ be a development graph. For $N \in \mathcal{N}$, the* **theory** *$Th_{\mathcal{S}}(N)$ of N wrt. a development graph $\mathcal{S}$ is defined by*

$$Th_{\mathcal{S}}(N) \quad = \quad \left[\Phi^N \cup \bigcup_{K \xrightarrow{\sigma} N \in \mathcal{S}} \sigma(Th_{\mathcal{S}}(K)) \cup \bigcup_{K \xrightarrow{\sigma} N \in \mathcal{S}} \sigma(\Phi^K) \right]^{\vdash_{\Sigma^N}}$$

For flatable nodes N, $Th_{\mathcal{S}}(N)$ captures N completely. However, this is not the case for nodes that are not flatable. Therefore, we cannot define a proof-theoretic semantics of development graphs as in [AHMS00]. Rather, we have to use a model-theoretic semantics. In Sect. 5 we will show how this model-theoretic semantics relates to the proof theoretic semantics given in [AHMS00].

Definition 8. *Given a node $N \in \mathcal{N}$, its associated class $\mathbf{Mod}_S(N)$ of models (or N-models for short) consists of those Σ^N-models n for which*

- *n satisfies the local axioms Φ^N,*
- *for each $K \xrightarrow{\sigma} N \in \mathcal{S}$, $n|_\sigma$ is a K-model,*
- *for each $K \xrightarrow{\sigma} N \in \mathcal{S}$, $n|_\sigma$ satisfies the local axioms Φ^K, and*
- *for each $K \xrightarrow[h]{\sigma} N \in \mathcal{S}$, n has a σ-expansion k (i.e. $k|_\sigma = n$) that is a K-model.*

Complementary to definition links, which *define* the theories of related nodes, we introduce the notion of a *theorem link* with the help of which we are able to *postulate* relations between different theories. Theorem links are the central data structure to represent proof obligations arising in formal developments. Again we distinguish between local and global theorem links (denoted by $N \overset{\sigma}{-\!\!\blacktriangleright} M$ and $N \overset{\sigma}{-\!\!\rightarrow} M$ respectively). We also need theorem links $N \overset{\sigma}{-\!\!\blacktriangleright} \theta M$ (where for some Σ, $\theta \colon \Sigma \longrightarrow \Sigma^N$ and $\sigma \colon \Sigma \longrightarrow \Sigma^M$) involving hiding. The semantics of theorem links is given by the next definition.

Definition 9. *Let $\mathcal{S}$ be a development graph and N, M nodes in $\mathcal{S}$.*

$\mathcal{S}$ **implies** *a global theorem link $N \overset{\sigma}{-\!\!\blacktriangleright} M$ (denoted $\mathcal{S} \models N \overset{\sigma}{-\!\!\blacktriangleright} M$) iff for all $m \in \mathbf{Mod}_S(M)$, $m|_\sigma \in \mathbf{Mod}_S(N)$.*

$\mathcal{S}$ **implies** *a local theorem link $N \overset{\sigma}{-\!\!\rightarrow} M$ (denoted $\mathcal{S} \vdash N \overset{\sigma}{-\!\!\rightarrow} M$) iff for all $m \in \mathbf{Mod}_S(M)$, $m|_\sigma \models \phi$ for all $\phi \in \Phi^N$.*

$\mathcal{S}$ **implies** *a hiding theorem link $N \overset{\sigma}{-\!\!\blacktriangleright} \theta M$ (denoted $\mathcal{S} \models N \overset{\sigma}{-\!\!\blacktriangleright} \theta M$) iff for all $m \in \mathbf{Mod}_S(M)$, $m|_\sigma$ has a θ-expansion to some N-model.*

E.g. consider the development graph of the running example (cf. Fig. 1): The theorem link from SORTING to INSERTSORT expresses the postulation that the latter is a refinement of the former. Furthermore, common proof obligations in a formal development can be encoded into properties that specific global theorem links are implied by the actual development graph.

A global definition link $M \xrightarrow{\sigma} N$ in a development graph is a *conservative extension*, if every M-model can be expanded along σ to an N-model. We will allow to annotate a global definition link as $M \xrightarrow[c]{\sigma} N$, which shall express that it is a conservative extension. These annotations can be seen as another kind of proof obligations.

5 Rules for Development Graphs with Hiding

The rules for theorem proving in development graphs given in [AHMS00] allow to decompose a global theorem link into local theorem links. Unfortunately, it is not possible to decompose global theorem links starting from nodes with hiding definition links going (directly or indirectly) into them. This is because hiding is some kind of existential quantification, and in general it is not possible to decompose an existential quantification of a conjunction into existential quantifications of the conjuncts.

We therefore have to extend the set of rules from [AHMS00] to deal with hiding. We have two kinds of rules:

1. *Rules for hiding and conservative extension.* These rules are suited to push theorem links along the hidings inside the development graph, such that they eventually can be decomposed into local theorem links.
2. *Decomposition rules* from [AHMS00]. They allow to split global theorem links into local and hiding theorem links.

5.1 Rules for Hiding and Conservative Extension

We now come to the rules for hiding and conservative extension. We introduce two rules to shift theorem links over hiding, one dealing with hiding links on the left hand side of a theorem link, and the other one with hiding links on the right hand side of a theorem link.

Since the first rule is quite powerful, we need some preliminary notions. Given a node N in a development graph $\mathcal{S} = \langle \mathcal{N}, \mathcal{L} \rangle$, the idea is that we unfold the subgraph below N into a tree and form a diagram with this tree. More formally, define the *diagram* $D\colon I \longrightarrow \mathbf{Sign}$ *associated with* N together with a map $G\colon |I| \longrightarrow \mathcal{N}$ inductively as follows:

- $\langle N \rangle$ is an object in I, with $D(\langle N \rangle) = N^{\Sigma}$. Let $G(\langle N \rangle)$ be just N.
- if $i = \langle\, M \xrightarrow{l_1} \cdots \xrightarrow{l_n} N \,\rangle$ is an object in I with $l_1, \ldots, l_n$ non-local links in $\mathcal{L}$, and $l = K \xrightarrow{\sigma} M$ is a (global or local) definition link in $\mathcal{L}$, then

$$j = \langle\, K \xrightarrow{l} M \xrightarrow{l_1} \cdots \xrightarrow{l_n} N \,\rangle$$

is an object in I with $D(j) = \Sigma^K$, and l is a morphism from j to i in I with $D(l) = \sigma$. We set $G(j) = K$.

– if $i = \langle\, M \xrightarrow{l_1} \cdots \xrightarrow{l_n} N \,\rangle$ is an object in I with $l_1, \ldots, l_n$ non-local links in $\mathcal{L}$, and $l = K \xrightarrow[h]{\sigma} M$ is a hiding link in $\mathcal{L}$, then

$$j = \langle\, K \xrightarrow{l} M \xrightarrow{l_1} \cdots \xrightarrow{l_n} N \,\rangle$$

is an object in I with $D(j) = \Sigma^K$, and l is a morphism from i to j in I with $D(l) = \sigma$. We set $G(j) = K$.

This means that the graph is just unfolded to the diagram. The unfolding is necessary to achieve that in the diagram there is a distinction between instances of the same node that are imported via different paths into another node.

Theorem-Hide-Shift. This rule (cf. Fig. 2) is used if a hiding link occurs on the right-hand side of a theorem link. For this rule $D\colon I \longrightarrow \mathbf{Sign}$ is the diagram associated with N, $\mu_i\colon D(i) \longrightarrow Colim\ D$ are the colimit injections ($i \in |I|$), C is a new isolated node with signature $Colim\ D$, and with ingoing global definition links $G(i) \xrightarrow{\mu_i} C$ for $i \in |I|$. Here, an isolated node is one with no local axioms and no ingoing definition links other than those shown in the rule

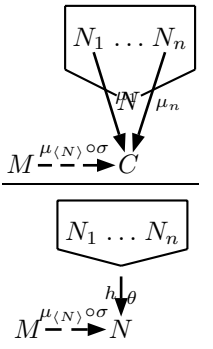
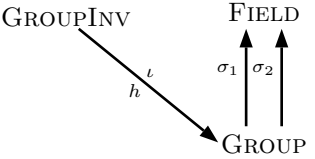
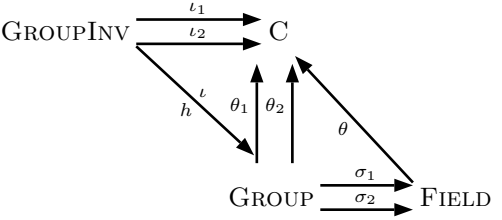


Fig. 2. Rule *Theorem-Hide-Shift*

We now illustrate why the unfolding of the subgraph under N in the rule *Theorem-Hide-Shift* is needed. Consider the development graph



defining groups with the help of groups with inverse (by hiding the inverse) and then defining fields using groups twice: both for the additive and the multiplicative group. If we would take the colimit of this graph, we would identify the additive with the multiplicative group in *Field*. This is not what we want. The unfolding of the rule *Theorem-Hide-Shift* now doubles the signature of groups and groups with inverse, leading to a signature $Colim\ D$ containing the additive and the multiplicative group, and an additive and a multiplicative inverse. The graph for the premise of the rule is then



In the node C, one can reason about both inverses in parallel, while the theory of groups with inverse is not doubled (as it would be the case with approaches that flatten specifications).

Hide-Theorem-Shift. This rule (cf. Fig. 3) replaces hiding theorem links by normal theorem links. This is only possible if on the right-hand side of the hiding theorem link, a conservative definition link

occurs, and furthermore $\sigma' \circ \theta = \theta' \circ \sigma$.

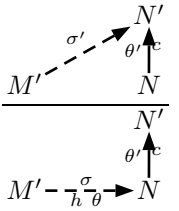


Fig. 3. Rule *Hide-Theorem-Shift*

Cons-Shift. The previous rule to replace hiding theorem links requires conservative definition links. In order to be able to derive new conservative definition links from existing ones, we introduce a rule which allows their derivation. For this rule (cf. Fig. 4) we must require that

$$\begin{array}{ccc} \Sigma^M & \xrightarrow{\sigma} & \Sigma^N \\ \downarrow{\scriptstyle \theta} & & \downarrow{\scriptstyle \theta'} \\ \Sigma^{M'} & \xrightarrow{\sigma'} & \Sigma^{N'} \end{array}$$

is a pushout, and moreover, that N' is isolated.

In addition to the above rules, one would use logic-specific rules for syntactically determining conservative extensions (e.g. extensions by definitions).

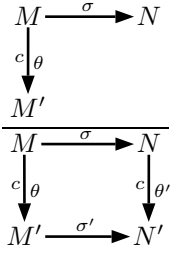


Fig. 4. Rule *Cons-shift*

Proposition 1. *The above rules are sound.*

Proof. Theorem-Hide-Shift: Assume that $\mathcal{S} \models M \xrightarrow{\mu_{\langle N \rangle} \circ \sigma} C$. Let n be an N-model. We have to show $n|_\sigma$ to be an M-model in order to establish the holding of $M \xrightarrow{\sigma} N$. We inductively define a family $(m_i)_{i \in |I|}$ of models $m_i \in \mathbf{Mod}(G(i))$ by putting

- $m_{\langle N \rangle} := n$,
- $m_{\langle K \xrightarrow{l} M \xrightarrow{l_1} \cdots \xrightarrow{l_n} N \rangle} := m|_\sigma$, where $l = K \xrightarrow{\sigma} M$ and $m = m_{\langle M \xrightarrow{l_1} \cdots \xrightarrow{l_n} N \rangle}$, and
- $m_{\langle K \xrightarrow{l} M \xrightarrow{l_1} \cdots \xrightarrow{l_n} N \rangle}$ is a σ-expansion of m to a K-model, where $l = K \xrightarrow[h]{\sigma} M$ and $m = m_{\langle M \xrightarrow{l_1} \cdots \xrightarrow{l_n} N \rangle}$.

It is easy to show that this family is consistent with D. By weak amalgamation, there is a $\Sigma^C = Colim\ D$-model c with $c|_{\mu_i} = m_i$. The latter implies that c is a C-model. By the assumption, $c|_{\mu_{\langle N \rangle} \circ \sigma} = m_{\langle N \rangle}|_\sigma = n|_\sigma$ is an M-model.

Hide-Theorem-shift: Assume that $\mathcal{S} \models M' \xrightarrow{\sigma'} N'$ and $N \xrightarrow[c]{\theta'} N'$ is conservative. We have to show that $\mathcal{S} \models M' \xrightarrow{\sigma} \theta N$. Let n be an N-model. Since $N \xrightarrow[c]{\theta'} N'$ is conservative, n can be expanded to an N'-model n' with $n'|_{\theta'} = n$. By

the assumption, $n'|_{\sigma'}$ is an M'-model. Thus, $n'|_{\sigma' \circ \theta} = n'|_{\theta' \circ \sigma} = n|_\sigma$ has a θ-expansion to an M'-model.

Cons-shift: Assume that $M \xrightarrow[c]{\theta} M'$ is conservative. We have to prove that $N \xrightarrow[c]{\theta'} N'$ is conservative as well. Let n be an N-model. Since $M \xrightarrow{\theta} M'$ is conservative, $n|_\sigma$ has a θ-expansion m' being an M'-model. By weak amalgamation, there is some $\Sigma^{N'}$-model n' with $n'|_{\sigma'} = m'$ and $n'|_{\theta'} = n$. Since N' is isolated, n' is an N'-model.

5.2 Rules for Decomposition

The rules for decomposition are taken from [AHMS00]. The rule

Glob-Decomposition has to be changed. It now

decomposes a global theorem link from N to M into local theorem links into M from those nodes from which N is reachable and into hiding theorem links into M from those nodes which are the source of a hiding link going into some node from which N is reachable.

Moreover, we have added a subsumption rule, stating that global reachability suffices to establish a global theorem link.

Glob-Decomposition:

$$\frac{\bigcup_{K \xrightarrow{\sigma'} N} \{K \xrightarrow{\sigma \circ \sigma'} M\} \cup \bigcup_{L \xrightarrow{\theta} K \text{ and } K \xrightarrow{\sigma'} N} \{L \xrightarrow{\sigma \circ \sigma'} \theta M\}}{N \xrightarrow{\sigma} M}$$

Subsumption:

$$\frac{N \xrightarrow{\sigma} M}{N \xrightarrow{\sigma} M}$$

Since the other rules have not changed, we just recall them here for sake of completeness.

Loc-Decomposition I:

$$\frac{K \xrightarrow{\sigma} L}{K \xrightarrow{\sigma''} M} \text{ if } L \xrightarrow{\sigma'} M \text{ and } \sigma''(\Phi(K)) = \sigma'(\sigma(\Phi(K)))$$

Loc-Decomposition II:

$$\frac{}{N \xrightarrow{\sigma} M} \text{ if } N \xrightarrow{\sigma'} M \text{ and } \sigma(\Phi(N)) = \sigma'(\Phi(N))$$

Basic Inference:

$$\frac{Th_{\mathcal{S}}(M) \vdash \sigma(\phi) \text{ for each } \phi \in \Phi^N}{N \xrightarrow{\sigma} M}$$

Given a development graph containing some theorem links that have to be proved, the intention is that the above rules will be applied in a backwards manner as long as possible. The rules for hiding allow to shift the (global) theorem links to nodes without hiding involved, while the rules for decomposition allow to decompose the global theorem links into local ones. With *Basic Inference*, local theorem links can be proved using the entailment relation of the base logic $\mathcal{LOG}$.

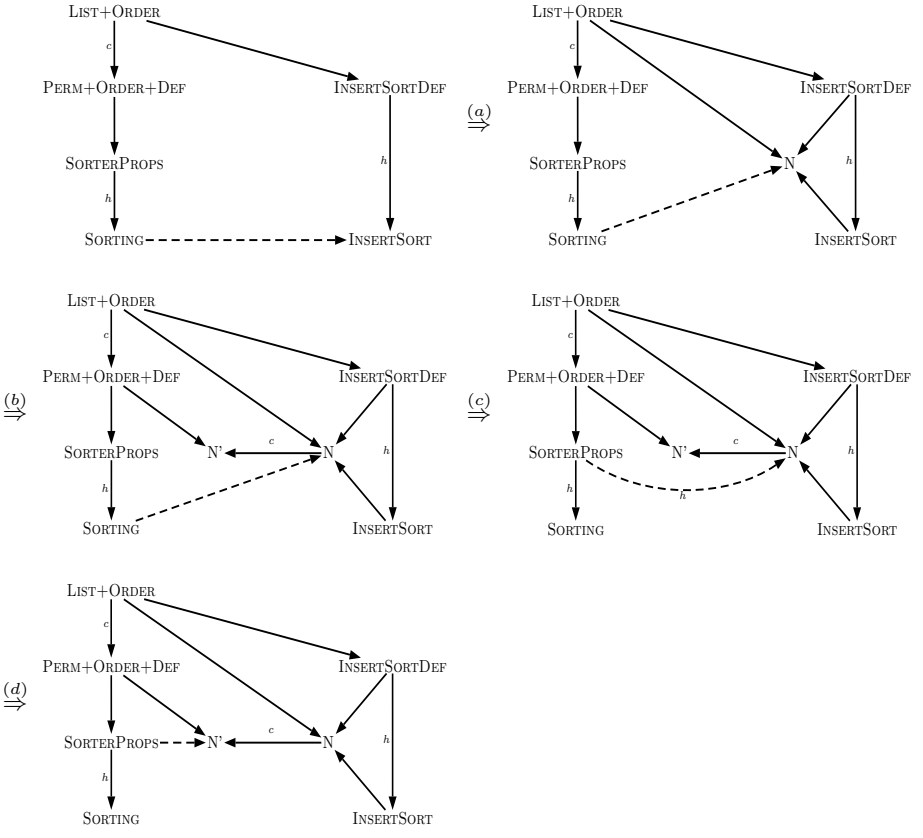


Fig. 5. Reduction of theorem links to *flatable* nodes in the running example.

6 Example

We now demonstrate the (backward manner) use of the rules with the example development graph from Sect. 4. The goal is to reduce the theorem link between Sorting and InsertSort to theorem links between flatable nodes. The derivation is shown in Fig. 5. In the first step (a) the *Theorem-Hide-Shift* rule is applied, which introduces the new node N and the new global definition links. In the second step (b), we infer conservative relationships by applying the rule *Cons-Shift*. This introduces the new node N' and the respective global definition links. Now the theorem link can be reduced to a hiding theorem link from SorterProps to N by *Glob-Decomposition* (step (c)). Finally, this hiding theorem link can be reduced to the theorem link between SorterProps and N' using the rule *Hide-Theorem-Shift*. Since both SorterProps and N' are flatable, one now can use reasoning in the logic to prove the remaining theorem link (via *Basic Inference*).

7 Results about Completeness

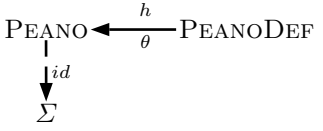
The soundness of our rules is established by proposition 1 for the hiding rules, while showing soundness of the other rules is easy. Another question is the completeness of our rules. We have the following counterexample:

Proposition 2. *Let FOL be the usual first-order logic with a recursively axiomatized complete entailment system. Solving the question whether a global theorem link holds in a development graph with hiding over FOL is not recursively enumerable. Thus, any recursively axiomatized calculus for development graphs with hiding is incomplete.*

Proof. This can be seen as follows. Let Σ be the FOL-signature with a sort *nat* and operations for zero and successor, addition and multiplication and take the usual second-order Peano axioms characterizing the natural numbers uniquely up to isomorphism, plus the defining axioms for addition and multiplication. Without loss of generality, we can assume that these axioms are combined into a single axiom of the form

$$\forall P\colon pred(nat)\ .\ \varphi$$

where φ is a first-order formula. Let ψ be any sentence over Σ. Let $\theta\colon \Sigma \longrightarrow \Sigma'$ add a predicate $P : pred(nat)$ to Σ. Consider the development graph



where Σ and PEANO are nodes with signature Σ and no local axioms, whereas PEANODEF is a node with signature Σ' and local axiom $\varphi \Rightarrow \psi$.

Now we have that PEANO $\xrightarrow{id}$ Σ holds iff each $\Sigma - model$ has a PEANODEF-expansion. It is easy to see that this holds iff the second-order formula $\exists P : pred(nat).\varphi \Rightarrow \psi$ is valid. By the quantifier rules for prenex normal form, this is equivalent to $\forall P : pred(nat).\varphi \models \psi$, i.e. equivalent to the fact that ψ holds in the second-order axiomatization of Peano arithmetic. By Gödels incompleteness theorem, this question is not recursively enumerable. □

In spite of this negative result, there is still the question of a relative completeness w.r.t. a given oracle deciding conservative extensions. Such a completeness result has been proved by Borzyszkowski [Bor01] in a similar setting. We are going to prove a similar result here. We first need a preparatory lemma:

Lemma 1. *If C is a flatable node or if C is constructed as in the rule Theorem-Hide-Shift, then any Σ^C-model satisfying $Th_{\mathcal{S}}(C)$ is already a C-model.*

Proof. If C is flatable, the result follows by an easy induction over the definition links indirectly or directly going into C. Now let C be constructed as

in the rule *Theorem-Hide-Shift*. We use the notation introduced in connection with the construction of C. For $i \in |I|$, let $len(i)$ be the length of the path i, and let p be the maximum of all $len(i)$, $i \in |I|$. We prove by induction over $p - len(i)$ that for all $i \in |I|$, $c|_{\mu_i}$ is a $G(i)$-model. Since C contains no local axioms, the result then follows. Let $i = \langle\, M \xrightarrow{l_1} \cdots \xrightarrow{l_n} N \,\rangle \in |I|$. By induction hypothesis, for each ingoing link $K \xrightarrow{l} M$, $c|_{\mu_j}$ is a K-model for $j = \langle\, K \xrightarrow{l} M \xrightarrow{l_1} \cdots \xrightarrow{l_n} N \,\rangle$. Now if $l = K \xrightarrow{\sigma} M$, $c|_{\mu_i \circ \sigma} = c|_{\mu_j}$, while if $l = K \xrightarrow[h]{\sigma} M$, $c|_{\mu_j \circ \sigma} = c|_{\mu_i}$. In the former case, $c|_{\mu_i}$ reduces to a K-model, while in the latter case, $c|_{\mu_i}$ expands to a K-model. Hence in both cases, the link l is satisfied by $c|_{\mu_i}$. Since c satisfies $Th_{\mathcal{S}}(C)$ and $N = G(i) \xrightarrow{\mu_i} C$, $c|_{\mu_i}$ also satisfies the local axioms in M. Hence, $c|_{\mu_i}$ is a model of $M = G(i)$. □

Theorem 1. *Assume that the underlying logic $\mathcal{LOG}$ is complete. Then the rule system for development graphs with hiding is complete relative to an oracle for conservative extensions.*

Proof. Assume $\mathcal{S} \models M \overset{\sigma}{-\!\!\rightarrow} N$. We show that there is some faithful extension $\mathcal{S}_1$ of $\mathcal{S}$ (i.e. new nodes and new definition links are added, but the latter go only into new nodes) such that $\mathcal{S}_1 \vdash M \overset{\sigma}{-\!\!\rightarrow} N$.

Let $D\colon I \longrightarrow \mathbf{Sign}$ and C be as in the rule *Theorem-Hide-Shift*, and let c be a Σ^C-model satisfying $Th_{\mathcal{S}}(C)$. By Lemma 1, c is a C-model. Hence, $c|_{\mu_{\langle N \rangle}}$ is an N-model, and by the assumption $\mathcal{S} \models M \overset{\sigma}{-\!\!\rightarrow} N$, $c|_{\mu_{\langle N \rangle} \circ \sigma}$ is an M-model. We now have for any $K \overset{\theta}{-\!\!\rightarrow} M$:

1. $c|_{\mu_{\langle N \rangle} \circ \sigma \circ \theta} \models \Phi^K$. By the satisfaction condition for institutions, we get $c \models \mu_{\langle N \rangle}(\sigma(\theta(\Phi^K)))$. By completeness of the underlying logic, we get $Th_{\mathcal{S}}(C) \vdash \mu_{\langle N \rangle}(\sigma(\theta(\Phi^K)))$. By *Basic Inference*, $\mathcal{S} \vdash K \overset{\mu_{\langle N \rangle} \circ \sigma \circ \theta}{-\!\!\rightarrow} C$.
2. For $L \xrightarrow[h]{\tau} K$, form the pushout

$$\begin{array}{ccc} \Sigma^K & \xrightarrow{\mu_{\langle N \rangle} \circ \sigma \circ \theta} & \Sigma^C \\ \downarrow{\scriptstyle \tau} & & \downarrow{\scriptstyle \tau'} \\ \Sigma^L & \xrightarrow{\mu'} & \Sigma' \end{array}$$

   and obtain a new development graph $\mathcal{S}'$ from $\mathcal{S}$ by introducing a new node L' with signature Σ' and two ingoing definition links $L \xrightarrow{\mu'} L'$ and $C \xrightarrow{\tau'} L'$. The latter link is conservative: for any C-model c_1, $c_1|_{\mu_{\langle N \rangle} \circ \sigma \circ \theta}$ has a τ-expansion to an L-model c_2, and by weak amalgamation, there is some Σ'-model c_3 with $c_3|_{\tau'} = c_1$ and $c_3|_{\mu'} = c_2$, which is hence an L'-model. By the oracle for conservativity, we get $C \xrightarrow[c]{\tau'} L'$. Now $\mathcal{S}' \vdash L \overset{\mu'}{-\!\!\rightarrow} L'$ by *Subsumption*. By *Hide-Theorem-Shift*, we get $\mathcal{S}' \vdash L \overset{\mu_{\langle N \rangle} \circ \sigma \circ \theta}{-\!\!\rightarrow} \tau C$.

Let $\mathcal{S}_1$ be the union of all the $\mathcal{S}'$ constructed in step 2 above (assuming that all the added nodes are distinct). By *Glob-Decomposition*, we get $\mathcal{S}_1 \vdash M \overset{\mu_{\langle N \rangle} \circ \sigma}{-\!\!\rightarrow} C$. By *Theorem-Hide-Shift*, we get $\mathcal{S}_1 \vdash M \overset{\sigma}{-\!\!\rightarrow} N$. □

Corollary 1. *If $\mathcal{LOG}$ is complete, the decomposition rules are complete for proving theorem links between flatable nodes.*

Proof. By inspecting the proof of Theorem 1, one can see that for theorem links between flatable nodes, case 2 is never entered, and thus neither the rules *Cons-Shift* and *Hide-Theorem-Shift* nor the oracle for conservativeness are needed. Since Lemma 1 also holds for flatable nodes, one can replace the node C in the proof of Theorem 1 with the node N, thus also avoiding the use of *Theorem-Hide-Shift*. □

8 Conclusion and Related Work

We have extended the notion of development graph [AHMS00] to deal also with hiding. We have developed a semantics and a proof system for development graphs with hiding. The proof system can easily shown to be sound.

Concerning completeness, with a counterexample, we show that there can be no complete recursively axiomatized proof system for development graphs with hiding. However, we have shown our proof rules to be complete relative to a given oracle for detecting conservative extensions. We thus have achieved the same degree of completeness as Borzyszkowski [Bor01] rule system for structured specifications. In one sense our system is more complete than Borzyszkowski's: since our *Theorem-Hide-Shift* rule simulates something like Borzyszkowski's normal forms, we do not have to rely on the Craig interpolation property. For example, it is possible to solve a counterexample showing incompleteness in case of failure of interpolation in [Bor01] with our rules. Borzyszkowski refrains from doing normal form inference because with his way of computing normal forms, the structure of the specification is lost. Note that this is not the case with our rules, since they just extend the structure of the development graph, while the axioms are kept locally. One can even further optimize the rule *Theorem-Hide-Shift* by reducing the constructed diagram to those nodes that are really necessary, which can be achieved by taking the so-called final subcategory [AR94].

Compared with the rules in [Bor01], we have fewer but more complex rules. Our rules involve colimit computations that may be tedious for humans using the rules directly, but that are no problem for computer assisted proofs. Indeed, by exploiting the graph structure, development graphs with hiding can lead to much more efficient proofs than possible when using the usual proof rules for structured specifications as in [Bor01].

Moreover, we expect no difficulty when extending the management of change developed in [AHMS00] to the case of development graphs with hiding in order to integrate the presented approach into the development graph system of INKA 5.0 [AHMS99]. Then it will be possible to support the maintenance of changes in developments by machine also when hiding are present.

In [AHMS00], we have described a translation from CASL structured specifications to development graphs. It should be straightforward to extend this translation to structured specifications with hiding.

References

[AHMS00] S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an evolutionary formal software-development using CASL. In C. Choppy, D. Bert, and P. Mosses, (Eds.), *Recent Developments in Algebraic Development Techniques, 14th International Workshop, WADT'99*, Bonas, France, LNCS 1827, page 73–88, Springer-Verlag, 2000.

[AHMS99] S. Autexier, D. Hutter, H. Mantel, A. Schairer. System Description: Inka 5.0 - A Logic Voyager. In H. Ganzinger, (Ed.), *Proceedings of CADE-16*, Trento, Italy, LNAI 1632, Springer-Verlag, 1999.

[AR94] J. Adámek, J. Rosický. *Locally Presentable and Accessible Categories*, Cambridge University Press, 1994.

[Bor01] T. Borzyszkowski. Logical systems for structured specifications. *Theoretical Computer Science*. To appear.

[CASL98] CoFI Language Design Task Group. *The Common Algebraic Specification Language (*Casl*) – Summary*, Version 1.0 and additional Note S-9 on Semantics, available from `http://www.brics.dk/Projects/CoFI`, 1998.

[CM97] M. Cerioli, J. Meseguer. May I borrow your logic?, *Theoretical Computer Science*, 173:311–347, 1997.

[DGS91] R. Diaconescu, J. Goguen, P. Stefaneas. Logical support for modularization, In G. Huet, G. Plotkin, (Eds), *Workshop on Logical Frameworks*, 1991.

[GB92] J. A. Goguen and R. M. Burstall: Institutions: Abstract model theory for specification and programming, *Journal of the Association for Computing Machinery*, 39:95–146, 1992. Predecessor in: LNCS 164, 221–256, 1984.

[HWB97] R. Hennicker, M. Wirsing, M. Bidoit. Proof systems for structured specifications with observability operators, *Theoretical Computer Science*, 173(2):393–443, 1997.

[HST94] R. Harper, D. Sannella, A. Tarlecki. Structured presentations and logic representations, In *Annals of Pure and Applied Logic*, 67:113-160, 1994.

[Mes89] J. Meseguer. General logics, In *Logic Colloquium 87*, pages 275–329, North Holland, 1989.

[MKK97] T. Mossakowski, Kolyang, B. Krieg-Brückner. Static semantic analysis and theorem proving for CASL, In *12th Workshop on Algebraic Development Techniques*, Tarquinia, LNCS 1376, pages 333–348, Springer-Verlag, 1998.

[ST88] D. Sannella, A. Tarlecki. Specifications in an arbitrary institution, *Information and Computation*, 76(2-3):165-210, 1988.

[ST92] D. Sannella, A. Tarlecki. Towards Formal Development of Programs from Algebraic Specifications: Model-Theoretic Foundations, *19th ICALP*, Springer, LNCS 623, pages 656–671, Springer-Verlag, 1992.

[SMHKT01] L. Schröder, T. Mossakowski, P. Hoffman, B. Klin, and A. Tarlecki. Semantics for architectural specifications in CASL, In H. Hußmann, (Ed.), *Proceedings of Fundamental Approaches to Software Engineering (FASE 2001)*, Genova, Italy, 2001.

[VSE96] D. Hutter et. al. Verification Support Environment (VSE), *Journal of High Integrity Systems*, Vol. 1, pages 523–530, 1996.

A Logic for the Java Modeling Language JML

Bart Jacobs and Erik Poll

Dept. Computer Science, Univ. Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.
{bart,erikpoll}@cs.kun.nl http://www.cs.kun.nl/~{bart,erikpoll}

Abstract. This paper describes a specialised logic for proving specifications in the Java Modeling Language (JML). JML is an interface specification language for Java. It allows assertions like invariants, constraints, pre- and post-conditions, and modifiable clauses as annotations to Java classes, in a design-by-contract style. Within the LOOP project at the University of Nijmegen JML is used for specification and verification of Java programs. A special compiler has been developed which translates Java classes together with their JML annotations into logical theories for a theorem prover (PVS or Isabelle). The logic for JML that will be described here consists of tailor-made proof rules in the higher order logic of the back-end theorem prover for verifying translated JML specifications. The rules efficiently combine partial and total correctness (like in Hoare logic) for all possible termination modes in Java, in a single correctness formula.

1 Introduction

JML (for Java Modeling Language) [15,14] is a specification language tailored to Java, primarily developed at Iowa State University. It allows assertions to be included in Java code, specifying for instance pre- and postconditions and invariants in the style of Eiffel and the design-by-contract approach [18]. JML has been integrated with the specification language used for ESC/Java, the extended static checker developed at Compaq System Research Center [17,27].

At Nijmegen, a formal denotational semantics has been developed for essentially all of sequential Java. A compiler has been built, the LOOP tool [5], which translates a Java program into logical theories describing its semantics [13,4,10, 9,12,8,26]. These logical theories are in a format that can serve as input for theorem provers, which can then be used to prove properties of the Java program, thus achieving a high level of reliability for this program. The LOOP tool supports output for the theorem provers PVS [20] and Isabelle [21]. This approach to verification of Java has demonstrated its usefulness for instance with the proof of a non-trivial invariant for the Vector class in the standard Java API [11]. The current main application area is JavaCard [1], see [24,25]. The LOOP tool is being extended to JML, so that it can be used to verify JML-annotated Java source code. At the moment this works for a kernel of JML.

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 284–299, 2001.

One advantage of using a formal specification language is that tool support becomes possible. Work on tool support for JML focuses on the generation of run-time checks on preconditions for testing, at Iowa State University [6] extended static checking, at Compaq System Research Center, and verification using the LOOP tool, at the University of Nijmegen. This offers a wide range of validation options—a key advantage of JML.

This paper presents a logic for reasoning about (sequential) Java programs which is the result of several years of experience in this area. The semantical and logical approach to Java within the LOOP project is bottom-up: it starts from an (automatic) translation of Java programs into what is ultimately a series of low level get- and put-operations on a suitable memory model [4]. From this point onwards, several steps have been taken up the abstraction ladder.

1. At first, the results to be proved (about the Java program under consideration) were formulated in the higher order logic of the back-end theorem prover (PVS or Isabelle), and proved by fully unpacking their meaning in terms of the low level (get and put) operations on the memory. Only relatively small programs can be handled like this, despite the usefulness of automatic rewriting.
2. Next a suitable Hoare logic for Java was introduced (in PVS and Isabelle) [10] for compositional reasoning about programs at a higher level of abstraction. This logic has different Hoare triples, corresponding to partial / total correctness for each of the possible termination modes of Java statements and expressions (normal / exception / return / break / continue). In theory this logic is appropriate, but in practice it involves too many rules and leads to too many duplications of proof obligations (for the different termination modes).
3. In a further abstraction step, the results to be proved were no longer formulated in PVS or Isabelle, but in a suitable specification language for Java, namely JML [14]. JML makes it possible to write specifications for Java programs without the need to know the details of these programs in PVS/Isabelle. Again, the translation from (a kernel of) JML to PVS/Isabelle is done automatically.
4. In a final step—the topic of this paper—a tailor-made logic is developed for proving (in PVS/Isabelle) these JML specifications. This logic involves syntax-driven rules (in PVS/Isabelle), supported by appropriate proof strategies, enabling the user to go step-by-step through a method body. The logic combines partial and total correctness together with the five different termination modes in a single correctness formula, resembling JML method specifications. This makes the logic both powerful and efficient in its use. Soundness of all these rules has been proved on the basis of the underlying semantics for Java. Most of the proofs are easy and just involve many case distinctions. The soundness of the while rule, see Subsection 5.6, is non-trivial.

The rules we describe below only handle the standard, imperative part of (sequential) Java, and not its typically object-oriented features (dealing for ex-

ample with dynamic binding), as in [22,19]. We do not need these rules because we can always fall back on our low level semantics where these issues are handled automatically [9]. This is a crucial point. Our logic for JML is not used directly at the Java source code level—as is standard in Hoare logics, see [3,16, 7,22]—but at the translated Java code in the back-end theorem prover, *i.e.* on the semantical level. But since the translation performed by the LOOP tool is compositional, there is not much difference: during proofs in the logic for JML one still follows the original code structurally. In a forward approach (following the execution order) one typically peels off the leading statements step-by-step, adapting the precondition in a suitable way. In every step one has to prove this adaptation of the precondition, as a result of the leading statement. In our approach the latter is typically done *without* the logic for JML, by going down to the lowest semantical level (as in 1 above), making efficient use of automatic rewriting. As mentioned, this works well for small programs. Note that an important consequence of working at the semantic level

This combination of high level proof rules and low level automatic rewriting on the basis of the underlying semantics forms the strength of our pragmatic approach, where we only introduce logical rules when this really suits us, in order to achieve a higher level of abstraction in proofs. A consequence of working at the semantical level is that we cannot really define a notion of completeness for our higher level rules (like in [19]), because completeness can only be defined for a syntactic level w.r.t. some lower semantic level.

In this paper we shall only talk about proving JML specifications for certain Java implementations. We shall not use this here, but in certain cases these proofs may actually rely on other JML specifications, for example for methods which are native (implemented in some other language than Java), or which may be overridden. In the latter case one cannot rely on a specific implementation, because it may be different in subclasses. In a behavioural approach to subtyping [2] (see also [23]) one then assumes that all implementations in subclasses satisfy the specification in the class in which the method is first introduced. This specification will form the basis for verifications.

In order to explain our logic for JML, the paper will have to introduce quite a few languages: Java and its JML annotations (Section 2), higher order logic (as used in PVS and Isabelle) and the representation of Java statements and expressions therein (Section 3), the meaning of JML method specifications in logic (Section 4), and finally the rules themselves. Necessarily we cannot describe all details, and are forced to concentrate on the essentials. The paper involves an example specification in JML, verified in PVS using the logic for JML. It is the same example as in [10]—this time not on abstraction level 2 but on level 4, as described above.

2 Class and Method Specifications in JML

This section gives a brief impression of JML, concentrating on method specifications. For more information, see [15,14]. JML adds assertions to Java by

writing them as special comments (`/*@ ... @*/` or `//@ ...`). These assertions are Java Boolean expressions extended with special operators, like `\forall`, `\exists`, `\result` or `\old(-)`. Classes can be enriched with invariants (predicates that should be preserved by all methods) or history constraints (relations that should hold between all pre- and post-states of all methods). Methods can be annotated with behaviour specifications which can be either `normal_behavior`, `exceptional_behavior` or simply `behavior`. The latter is typically used as follows for specifying a method `m`.

```
/*@ behavior
  @   diverges: <pre-condition for non-termination>
  @   requires: <precondition>
  @ modifiable: <items that can be modified>
  @    ensures: <postcondition for normal termination>
  @    signals: <postcondition for exceptional termination>
  @*/
void m() { ... }
```

Roughly, this says that if the precondition holds, then if the method `m` hangs / terminates normally / terminates abruptly, then the diverges / ensures / signals clause holds (respectively). When the `diverges` is `true` (resp. `false`) we have partial (resp. total) correctness. But note that when it is `false`, the method can still terminate abruptly. A `normal_behavior` (or `exceptional_behavior`) describes a situation where a method *must* terminate normally (or exceptionally), assuming that the precondition holds. For example, the class in Figure 1 contains an annotated method (from [10]) that searches for a certain pattern in an array using a single while loop. It has a non-trivial postcondition.

3 Semantics of Java Statements and Expressions

This section introduces a denotational semantics of Java statements and expressions in higher order logic. This logic is a common abstraction of the logics used by PVS and Isabelle/HOL, and will be introduced as we proceed.

First, there is a complicated type OM, for object memory, with various get- and put-operations, see [4]. In this paper the internal structure of OM is not relevant. The type OM serves as our state space on which statements and expressions act, as functions $\mathsf{OM} \to \mathsf{StatResult}$ and $\mathsf{OM} \to \mathsf{ExprResult}[\alpha]$, for a suitable result type α. These result types are introduced as labeled coproduct (also called variant or sum) types:

$$\begin{array}{ll}
\mathsf{StatResult} : \mathsf{TYPE} \stackrel{\text{def}}{=} & \mathsf{ExprResult}[\alpha] : \mathsf{TYPE} \stackrel{\text{def}}{=} \\
\quad \{\mathsf{hang}\colon \mathsf{unit} & \quad \{\mathsf{hang}\colon \mathsf{unit} \\
\quad \mid \mathsf{norm}\colon \mathsf{OM} & \quad \mid \mathsf{norm}\colon [\mathsf{ns}\colon \mathsf{OM}, \mathsf{res}\colon \alpha] \\
\quad \mid \mathsf{abnorm}\colon \mathsf{StatAbn}\} & \quad \mid \mathsf{abnorm}\colon \mathsf{ExprAbn}\}
\end{array}$$

with labels hang, norm and abnorm corresponding to the three termination modes in Java: non-termination, normal termination and abrupt termination. Notice

```
class Pattern {

    int [] base, pattern;

    /*@ normal_behavior
      @   requires: base != null && pattern != null &&
      @             pattern.length <= base.length;
      @ modifiable: \nothing;
      @    ensures: ( /// pattern occurs;
      @             \result >= 0 &&
      @             \result <= base.length - pattern.length &&
      @               /// \result gives the start position
      @             (\forall (int i)
      @                 0 <= i && i < pattern.length
      @                    ==> pattern[i] == base[\result+i]) &&
      @               /// pattern does not occur earlier
      @             (\forall (int j)
      @                 0 <= j && j < \result
      @                    ==> (\exists (int i)
      @                            0 <= i && i < pattern.length
      @                               && pattern[i] != base[j+i])))
      @             ||
      @             ( /// pattern does not occur
      @             \result == -1 &&
      @             (\forall (int j)
      @                 0 <= j && j < base.length - pattern.length
      @                    ==> (\exists (int i)
      @                            0 <= i && i < pattern.length
      @                               && pattern[i] != base[j+i])));
      @*/
    int find_first_occurrence () {
        int p = 0, s = 0;
        while (true)
            if (p == pattern.length) return s;
            else if (s + p == base.length) return -1;
            else if (base[s + p] == pattern[p]) p++;
            else { p = 0; s++; }
    }
}
```

Fig. 1. A pattern search method in Java with JML annotation

that a normally termination expression returns both a state (incorporating the possible side-effect) and a result value. This is indicated by a labeled product (record) type $[\mathsf{ns}\colon \mathsf{OM}, \mathsf{res}\colon \alpha]$. The result types StatAbn and ExprAbn for abrupt termination are subdivided differently for statements and expressions:

$$\mathsf{StatAbn} : \mathsf{TYPE} \stackrel{\text{def}}{=} \{\, \mathsf{excp}\colon [\, \mathsf{es}\colon \mathsf{OM}, \mathsf{ex}\colon \mathsf{RefType}\,] \mid \mathsf{rtrn}\colon \mathsf{OM} \mid \mathsf{break}\colon [\, \mathsf{bs}\colon \mathsf{OM}, \mathsf{blab}\colon \mathsf{lift}[\mathsf{string}]\,] \mid \mathsf{cont}\colon [\, \mathsf{cs}\colon \mathsf{OM}, \mathsf{clab}\colon \mathsf{lift}[\mathsf{string}]\,]\,\}$$

$$\mathsf{ExprAbn} : \mathsf{TYPE} \stackrel{\text{def}}{=} [\, \mathsf{es}\colon \mathsf{OM}, \mathsf{ex}\colon \mathsf{RefType}\,]$$

The type RefType is used for references, containing either the null-reference or a pointer to a memory location. It describes the reference to an exception object, in case an exception is thrown. The lift type constructor adds a bottom element bot to an arbitrary type, and keeps all original elements a as $\mathsf{up}\, a$. It is used because `break` and `continue` statements in Java can be used both with and without label (represented as string).

On the basis of this representation of statements and expressions all language constructs from (sequential) Java are formalised in type theory (and used in the translation performed by the LOOP tool). For instance, the composition of two statements $s, t\colon \mathsf{OM} \to \mathsf{StatResult}$ is defined as:

$$(s\,;t) \stackrel{\text{def}}{=} \lambda x\colon \mathsf{OM}.\, \mathsf{CASES}\ s \cdot x\ \mathsf{OF}\ \{\ \mathsf{hang} \mapsto \mathsf{hang} \mid \mathsf{norm}\, y \mapsto t \cdot y \mid \mathsf{abnorm}\, a \mapsto \mathsf{abnorm}\, a\,\}$$

where $\cdot$ is used for function application, and CASES for pattern matching on the labels of the StatResult coproduct type. What is important to note is that if the statement s hangs or terminates abruptly, then so does the composition $s\,;t$.

There is no space to describe all these constructs in detail. We mention some of them that will be used later. Sometimes we need to execute an expression only for its side-effect (if any). This is done via the function E2S, defined as:

$$\begin{aligned}&\mathsf{E2S} \cdot e\ :\ \mathsf{OM} \to \mathsf{StatResult} \stackrel{\text{def}}{=}\\ &\quad \lambda x\colon \mathsf{OM}.\ \mathsf{CASES}\ e \cdot x\ \mathsf{OF}\ \{\\ &\qquad\qquad \mathsf{hang} \mapsto \mathsf{hang}\\ &\qquad\qquad \mid \mathsf{norm}\, y \mapsto \mathsf{norm}(y.\mathsf{ns})\\ &\qquad\qquad \mid \mathsf{abnorm}\, a \mapsto \mathsf{abnorm}(\mathsf{excp}(\mathsf{es} = a.\mathsf{es}, \mathsf{ex} = a.\mathsf{ex}))\,\}\end{aligned}$$

for $e\colon \mathsf{OM} \to \mathsf{ExprResult}[\alpha]$. The notation $y.\mathsf{ns}$ describes field selection associated with y in the labeled product $[\, \mathsf{ns}\colon \mathsf{OM}, \mathsf{res}\colon \alpha\,]$. In the last line an expression abnormality (an exception) is transformed into a statement abnormality. Java's `if-then-else` becomes:

$$\begin{aligned}&\mathsf{IF\text{-}THEN\text{-}ELSE} \cdot c \cdot s \cdot t\ :\ \mathsf{OM} \to \mathsf{StatResult} \stackrel{\text{def}}{=}\\ &\quad \lambda x\colon \mathsf{OM}.\ \mathsf{CASES}\ c \cdot x\ \mathsf{OF}\ \{\\ &\qquad\qquad \mathsf{hang} \mapsto \mathsf{hang}\\ &\qquad\qquad \mid \mathsf{norm}\, y \mapsto \mathsf{IF}\ y.\mathsf{res}\mathsf{THEN}\ s \cdot (y.\mathsf{ns})\mathsf{ELSE}\ t \cdot (y.\mathsf{ns})\\ &\qquad\qquad \mid \mathsf{abnorm}\, a \mapsto \mathsf{abnorm}(\mathsf{excp}(\mathsf{es} = a.\mathsf{es}, \mathsf{ex} = a.\mathsf{ex}))\,\}\end{aligned}$$

for $c\colon \mathsf{OM} \to \mathsf{ExprResult}[\mathsf{bool}]$ and $s, t\colon \mathsf{OM} \to \mathsf{StatResult}$. The formalisation of Java's `return` statement (without argument) is:

$$\mathsf{RETURN} : \mathsf{OM} \to \mathsf{StatResult} \stackrel{\text{def}}{=} \lambda x\colon \mathsf{OM}.\, \mathsf{abnorm}(\mathsf{rtrn}\, x)$$

This statement produces an abnormal "return" state. Such a return abnormality can be undone, via appropriate catch-return functions. In our translation of Java programs, such a function CATCH-RETURN is wrapped around every method body that returns `void`. First the method body is executed. This may result in an abnormal state, because of a return. In that case the function CATCH-RETURN turns the state back to normal again. Otherwise, it leaves everything unchanged.

$$\begin{array}{l}\mathsf{CATCH\text{-}RETURN} \cdot s : \mathsf{OM} \to \mathsf{StatResult}[\mathsf{OM}] \stackrel{\text{def}}{=} \\ \quad \lambda x\colon \mathsf{OM}.\ \mathsf{CASES}\ s \cdot x\ \mathsf{OF}\ \{ \\ \qquad\qquad \mathsf{hang} \mapsto \mathsf{hang} \\ \qquad\qquad \mid \mathsf{norm}\, y \mapsto \mathsf{norm}\, y \\ \qquad\qquad \mid \mathsf{abnorm}\, a \mapsto \mathsf{CASES}\ a\ \mathsf{OF}\ \{ \\ \qquad\qquad\qquad\qquad \mathsf{excp}\, e \mapsto \mathsf{abnorm}(\mathsf{excp}\, e) \\ \qquad\qquad\qquad\qquad \mid \mathsf{rtrn}\, z \mapsto \mathsf{norm}\, z \\ \qquad\qquad\qquad\qquad \mid \mathsf{break}\, b \mapsto \mathsf{abnorm}(\mathsf{break}\, b) \\ \qquad\qquad\qquad\qquad \mid \mathsf{cont}\, c \mapsto \mathsf{abnorm}(\mathsf{cont}\, c)\ \}\ \}\end{array}$$

The formalisation of creating and catching break and continue abnormalities works similarly, via function CATCH-BREAK and CATCH-CONTINUE.

4 Semantics of Method Specifications

To start we define two labeled product types incorporating appropriately typed predicates for the various termination modes of statements and expressions.

$$\begin{array}{l}\mathsf{StatBehaviorSpec} : \mathsf{TYPE} \stackrel{\text{def}}{=} \\ \quad [\ \mathsf{diverges}\colon \mathsf{OM} \to \mathsf{boolean}, \\ \quad\ \mathsf{requires}\colon \mathsf{OM} \to \mathsf{boolean}, \\ \quad\ \mathsf{statement}\colon \mathsf{OM} \to \mathsf{StatResult}, \\ \quad\ \mathsf{ensures}\colon \mathsf{OM} \to \mathsf{boolean}, \\ \quad\ \mathsf{signals}\colon \mathsf{OM} \to \mathsf{RefType} \to \mathsf{boolean}, \\ \quad\ \mathsf{return}\colon \mathsf{OM} \to \mathsf{boolean}, \\ \quad\ \mathsf{break}\colon \mathsf{OM} \to \mathsf{lift}[\mathsf{string}] \to \mathsf{boolean}, \\ \quad\ \mathsf{continue}\colon \mathsf{OM} \to \mathsf{lift}[\mathsf{string}] \to \mathsf{boolean}\ ]\end{array}$$

$$\begin{array}{l}\mathsf{ExprBehaviorSpec}[\alpha] : \mathsf{TYPE} \stackrel{\text{def}}{=} \\ \quad [\ \mathsf{diverges}\colon \mathsf{OM} \to \mathsf{boolean}, \\ \quad\ \mathsf{requires}\colon \mathsf{OM} \to \mathsf{boolean}, \\ \quad\ \mathsf{expression}\colon \mathsf{OM} \to \mathsf{ExprResult}[\alpha], \\ \quad\ \mathsf{ensures}\colon \mathsf{OM} \to \alpha \to \mathsf{boolean}, \\ \quad\ \mathsf{signals}\colon \mathsf{OM} \to \mathsf{RefType} \\ \qquad\qquad\qquad\qquad \to \mathsf{boolean}\ ]\end{array}$$

Notice that the StatBehaviorSpec type has more entries than ExprBehaviorSpec precisely because a statement in Java can terminate abruptly for more reasons than an expression.

There are associated predicates which give the "obvious" meaning.

$$
\begin{array}{l}
\mathsf{SB} \cdot sbs \ : \ \mathsf{boolean} \stackrel{\mathrm{def}}{=} \\
\quad \forall x \in \mathsf{OM}.\ sbs.\mathsf{requires} \cdot x \implies \\
\quad\quad \mathsf{CASES}\ sbs.\mathsf{statement} \cdot x\ \mathsf{OF}\ \{ \\
\quad\quad\quad \mathsf{hang} \shortmid \rightarrow sbs.\mathsf{diverges} \cdot x \\
\quad\quad\quad \mid \mathsf{norm}\, y \shortmid \rightarrow sbs.\mathsf{ensures} \cdot y \\
\quad\quad\quad \mid \mathsf{abnorm}\, a \shortmid \rightarrow \mathsf{CASES}\ a\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{excp}\, e \shortmid \rightarrow sbs.\mathsf{signals} \cdot (e.\mathsf{es}) \cdot (e.\mathsf{ex}) \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{rtrn}\, z \shortmid \rightarrow sbs.\mathsf{return} \cdot z \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{break}\, b \shortmid \rightarrow sbs.\mathsf{break} \cdot (b.\mathsf{bs}) \cdot (b.\mathsf{blab}) \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{cont}\, c \shortmid \rightarrow sbs.\mathsf{continue} \cdot (c.\mathsf{cs}) \cdot (c.\mathsf{clab})\ \}\ \} \\
\mathsf{EB} \cdot ebs \ : \ \mathsf{boolean} \stackrel{\mathrm{def}}{=} \\
\quad \forall x \in \mathsf{OM}.\ ebs.\mathsf{requires} \cdot x \implies \\
\quad\quad \mathsf{CASES}\ ebs.\mathsf{expression} \cdot x\ \mathsf{OF}\ \{ \\
\quad\quad\quad \mathsf{hang} \shortmid \rightarrow ebs.\mathsf{diverges} \cdot x \\
\quad\quad\quad \mid \mathsf{norm}\, y \shortmid \rightarrow ebs.\mathsf{ensures} \cdot (y.\mathsf{ns}) \cdot (y.\mathsf{res}) \\
\quad\quad\quad \mid \mathsf{abnorm}\, a \shortmid \rightarrow ebs.\mathsf{signals} \cdot (a.\mathsf{es}) \cdot (a.\mathsf{ex})\ \}
\end{array}
$$

for sbs: StatBehaviorSpec and ebs: ExprBehaviorSpec$[\alpha]$. Notice that the diverges predicate is evaluated in the pre-state, in case the statement/expression hangs, because in that case there is simply no post-state. All other predicates are evaluated in the post-state.

The LOOP compiler translates JML method specifications into elements of StatBehaviorSpec and ExprBehaviorSpec, depending on whether the method produces a result or not. The additional entries in StatBehaviorSpec which do not occur in JML specifications (the three last ones) are filled with default values. They may be filled with other values during proofs, typically because of catching of abnormalities, see Subsection 5.4.

For example, consider a JML method specification

```
/*@ behavior
  @   diverges: d;
  @   requires: p;
  @ modifiable: mod;
  @    ensures: q;
  @    signals: (E e) r;
  @*/
void m() { ... }
```

in a class with invariant `I`. This specification gets translated (by the LOOP compiler) into:

$$
\begin{aligned}
\forall z\colon \mathsf{OM}.\ \mathsf{SB} \cdot (\ & \mathsf{diverges} = [\![\mathtt{d}]\!], \\
& \mathsf{requires} = \lambda x\colon \mathsf{OM}.\ [\![\mathtt{I}]\!] \cdot x \wedge [\![\mathtt{p}]\!] \cdot x \wedge z = x, \\
& \mathsf{statement} = [\![\mathtt{m}]\!], \\
& \mathsf{ensures} = \lambda x\colon \mathsf{OM}.\ [\![\mathtt{I}]\!] \cdot x \wedge [\![\mathtt{q}]\!] \cdot x \cdot z \wedge z \approx_{\mathtt{mod}} x, \\
& \mathsf{signals} = \lambda x\colon \mathsf{OM}.\, \lambda a\colon \mathsf{RefType}.\ [\![\mathtt{I}]\!] \cdot x \wedge \\
& \qquad\qquad\qquad\qquad [\![a\ \mathtt{instanceof\ E}]\!] \wedge \\
& \qquad\qquad\qquad\qquad [\![\mathtt{r}]\!] \cdot x \cdot z \cdot a \wedge z \approx_{\mathtt{mod}} x, \\
& \mathsf{return} = \lambda x\colon \mathsf{OM}.\, \mathsf{false}, \\
& \mathsf{break} = \lambda x\colon \mathsf{OM}.\, \lambda l\colon \mathsf{lift}[\mathsf{string}].\, \mathsf{false}, \\
& \mathsf{continue} = \lambda x\colon \mathsf{OM}.\, \lambda l\colon \mathsf{lift}[\mathsf{string}].\, \mathsf{false}\ )
\end{aligned}
$$

The variable z is a logical variable which records the pre-state. It is needed because the normal and exceptional postconditions `q` and `r` may involve an operator `\old(e)`, requiring evaluation of `e` in the pre-state. The term $z \approx_{\mathtt{mod}} x$ is an appropriate translation of the modifiable clause, expressing that x and z are almost the same, except for the fields that are mentioned in the modifiable clause[1].

When translating a `normal_behavior` the diverges and signals fields are set to the constant predicate false; similarly, in an `exceptional_behavior` the diverges and ensures fields become false.

5 Rules for Proving Method Specifications

This section discusses a representative selection of the inference rules that are used for verifying JML method specifications. Some of these rules are bureaucratic, but most of them are "syntax driven". In a goal-oriented view they should be read up-side-down.

5.1 Diverges

Usually, the `diverges` clause in JML is constant, *i.e.* either true or false. Some of the rules below—for example, the composition rule in Figure 3—actually require it to be constant. This can always be enforced via the following rule—at the expense of duplication of the number of proof obligations, see Figure 2.

We illustrate the soundness of this rule. We assume therefore that the assumptions above the line hold. In order to prove the conclusion, we have to distinguish three main cases, for an arbitrary state $x\colon \mathsf{OM}$, satisfying $p \cdot x$:

- $s \cdot x$ hangs. According to the definition of SB, we have to prove $d \cdot x$. But $\neg d \cdot x$, leads to false by the second assumption.

[1] Often it is convenient to weaken the precondition to $\lambda x\colon \mathsf{OM}.\ [\![\mathtt{I}]\!] \cdot x \wedge [\![\mathtt{p}]\!] \cdot x \wedge z \approx_{\mathtt{mod}} x$, to obtain a more symmetric correctness formula.

$$\frac{\begin{array}{l}\mathsf{SB}\cdot(\ \mathsf{diverges}=\lambda x\colon \mathsf{OM}.\ \mathsf{true},\\ \quad\mathsf{requires}=\lambda x\colon \mathsf{OM}.\ p\cdot x\wedge d\cdot x,\\ \quad\mathsf{statement}=s,\\ \quad\mathsf{ensures}=q,\\ \quad\mathsf{signals}=r,\\ \quad\mathsf{return}=R,\\ \quad\mathsf{break}=B,\\ \quad\mathsf{continue}=C\ )\end{array}\qquad\begin{array}{l}\mathsf{SB}\cdot(\ \mathsf{diverges}=\lambda x\colon \mathsf{OM}.\ \mathsf{false},\\ \quad\mathsf{requires}=\lambda x\colon \mathsf{OM}.\ p\cdot x\wedge \neg d\cdot x,\\ \quad\mathsf{statement}=s,\\ \quad\mathsf{ensures}=q,\\ \quad\mathsf{signals}=r,\\ \quad\mathsf{return}=R,\\ \quad\mathsf{break}=B,\\ \quad\mathsf{continue}=C\ )\end{array}}{\begin{array}{l}\mathsf{SB}\cdot(\ \mathsf{diverges}=d,\\ \quad\mathsf{requires}=p,\\ \quad\mathsf{statement}=s,\\ \quad\mathsf{ensures}=q,\\ \quad\mathsf{signals}=r,\\ \quad\mathsf{return}=R,\\ \quad\mathsf{break}=B,\\ \quad\mathsf{continue}=C\ )\end{array}}$$

Fig. 2. Rule to force diverges predicates to be constant

- $s \cdot x$ terminates normally. The normal postcondition q follows in both cases $d \cdot x$ and $\neg d \cdot x$ from both the assumptions.
- $s \cdot x$ terminates abruptly. Similarly, one gets the appropriate postcondition from both the assumptions.

The soundness of most of the rules below (except for while) is similarly easy. Soundness of all the rules has been proved in PVS.

5.2 Composition

The rule that is most often used is the composition rule. It makes it possible to step through a piece of code by handling single statements one at a time, by introducing appropriate intermediate conditions, namely the p_1 in Figure 3.

$$\frac{\begin{array}{l}\mathsf{SB}\cdot(\ \mathsf{diverges}=\lambda x\colon \mathsf{OM}.\ b,\\ \quad\mathsf{requires}=p,\\ \quad\mathsf{statement}=s_1,\\ \quad\mathsf{ensures}=p_1,\\ \quad\mathsf{signals}=r,\\ \quad\mathsf{return}=R,\\ \quad\mathsf{break}=B,\\ \quad\mathsf{continue}=C\ )\end{array}\qquad\begin{array}{l}\mathsf{SB}\cdot(\ \mathsf{diverges}=\lambda x\colon \mathsf{OM}.\ b,\\ \quad\mathsf{requires}=p_1,\\ \quad\mathsf{statement}=s_2,\\ \quad\mathsf{ensures}=q,\\ \quad\mathsf{signals}=r,\\ \quad\mathsf{return}=R,\\ \quad\mathsf{break}=B,\\ \quad\mathsf{continue}=C\ )\end{array}}{\begin{array}{l}\mathsf{SB}\cdot(\ \mathsf{diverges}=\lambda x\colon \mathsf{OM}.\ b,\\ \quad\mathsf{requires}=p,\\ \quad\mathsf{statement}=s_1\,;s_2,\\ \quad\mathsf{ensures}=q,\\ \quad\mathsf{signals}=r,\\ \quad\mathsf{return}=R,\\ \quad\mathsf{break}=B,\\ \quad\mathsf{continue}=C\ )\end{array}}$$

Fig. 3. Composition rule

A special case of this rule which is often useful in practice has the intermediate condition p_1 of the form $\lambda x\colon \mathsf{OM}.\, p \cdot x \wedge p_2 \cdot x$, where p is the precondition of the goal, and p_2 is an addition to the precondition which holds because of the first statement s_1.

5.3 Return

Recall from Section 3 that the RETURN statement immediately terminates abruptly, by creating a "return" abnormality. The associated rule is much like a skip rule, see Figure 4.

5.4 Catching Returns

Recall that the LOOP compiler wraps a CATCH-RETURN function around each translated method body, in order to turn possible return abnormalities into normal termination. The associated rule in Figure 4 therefore puts the normal postcondition of the goal into the return position.

$$\frac{\forall x\colon \mathsf{OM}.\ p \cdot x \implies R \cdot x}{\begin{array}{l}\mathsf{SB} \cdot (\ \mathsf{diverges} = d,\\ \quad \mathsf{requires} = p,\\ \quad \mathsf{statement} = \mathsf{RETURN},\\ \quad \mathsf{ensures} = q,\\ \quad \mathsf{signals} = r,\\ \quad \mathsf{return} = R,\\ \quad \mathsf{break} = B,\\ \quad \mathsf{continue} = C\ )\end{array}} \qquad \frac{\begin{array}{l}\mathsf{SB} \cdot (\ \mathsf{diverges} = d,\\ \quad \mathsf{requires} = p,\\ \quad \mathsf{statement} = s,\\ \quad \mathsf{ensures} = q,\\ \quad \mathsf{signals} = r,\\ \quad \mathsf{return} = q,\\ \quad \mathsf{break} = B,\\ \quad \mathsf{continue} = C\ )\end{array}}{\begin{array}{l}\mathsf{SB} \cdot (\ \mathsf{diverges} = d,\\ \quad \mathsf{requires} = p,\\ \quad \mathsf{statement} = \mathsf{CATCH\text{-}RETURN} \cdot s,\\ \quad \mathsf{ensures} = q,\\ \quad \mathsf{signals} = r,\\ \quad \mathsf{return} = R,\\ \quad \mathsf{break} = B,\\ \quad \mathsf{continue} = C\ )\end{array}}$$

Fig. 4. Rules for the return and catch-return statements

Notice that via a rule like this an entry which is not used in JML specifications (namely return) can get a non-default value during proofs. This is the reason for including such additional entries in the definition of the type StatBehaviorSpec in Section 4.

5.5 If-Then-Else

Java has the `if-then` and `if-then-else` conditional statements. We only describe the relevant rule for the latter, see Figure 5. It deals with the possible side-effect and with the result of the condition c via the intermediate predicate qc.

$$\frac{\begin{array}{l}\mathsf{EB} \cdot (\ \mathsf{diverges} = \lambda x\colon \mathsf{OM}.\ b,\\ \quad \mathsf{requires} = p,\\ \quad \mathsf{expression} = c,\\ \quad \mathsf{ensures} = qc,\\ \quad \mathsf{signals} = r\ )\end{array} \qquad \begin{array}{l}\mathsf{SB} \cdot (\ \mathsf{diverges} = \lambda x\colon \mathsf{OM}.\ b,\\ \quad \mathsf{requires} = \lambda x\colon \mathsf{OM}.\\ \qquad qc \cdot x \cdot \mathsf{true}\\ \quad \mathsf{statement} = s_1,\\ \quad \mathsf{ensures} = q,\\ \quad \mathsf{signals} = r,\\ \quad \mathsf{return} = R,\\ \quad \mathsf{break} = B,\\ \quad \mathsf{continue} = C\ )\end{array} \qquad \begin{array}{l}\mathsf{SB} \cdot (\ \mathsf{diverges} = \lambda x\colon \mathsf{OM}.\ b,\\ \quad \mathsf{requires} = \lambda x\colon \mathsf{OM}.\\ \qquad qc \cdot x \cdot \mathsf{false}\\ \quad \mathsf{statement} = s_2,\\ \quad \mathsf{ensures} = q,\\ \quad \mathsf{signals} = r,\\ \quad \mathsf{return} = R,\\ \quad \mathsf{break} = B,\\ \quad \mathsf{continue} = C\ )\end{array}}{\begin{array}{l}\mathsf{SB} \cdot (\ \mathsf{diverges} = \lambda x\colon \mathsf{OM}.\ b,\\ \quad \mathsf{requires} = p,\\ \quad \mathsf{statement} = \mathsf{IF\text{-}THEN\text{-}ELSE} \cdot c \cdot s_1 \cdot s_2,\\ \quad \mathsf{ensures} = q,\\ \quad \mathsf{signals} = r,\\ \quad \mathsf{return} = R,\\ \quad \mathsf{break} = B,\\ \quad \mathsf{continue} = C\ )\end{array}}$$

Fig. 5. Rule for if-then-else

5.6 While

In a final rule, we consider Java's `while(c){s}` statement. It involves a condition `c` and a statement `s` which is iterated until the condition becomes false, or a form of abrupt termination arises. Especially, a `break` or `continue` statement, possibly with a label, may be used within a `while` statement (to jump out of the loop, or to jump to the next cycle). We refer to [10] for a detailed description of the formalisation of the `while` statement, and restrict ourselves to the relevant rule, see Figure 6.

$$\frac{\begin{array}{l}\mathsf{EB} \cdot (\ \mathsf{diverges} = \lambda x\colon \mathsf{OM}.\ b,\\ \quad \mathsf{requires} = \lambda x\colon \mathsf{OM}.\\ \qquad \mathsf{invariant} \cdot x \wedge\\ \qquad \mathsf{variant} \cdot x = a_1,\\ \quad \mathsf{expression} = c,\\ \quad \mathsf{ensures} = \lambda x\colon \mathsf{OM}.\ \lambda b\colon \mathsf{bool}.\\ \qquad \mathsf{IF}\ b\\ \qquad \mathsf{THEN}\ qc \cdot x \wedge \mathsf{variant} \cdot x = a_2\\ \qquad \mathsf{ELSE}\ q \cdot x,\\ \quad \mathsf{signals} = r\ )\end{array} \qquad \begin{array}{l}\mathsf{SB} \cdot (\ \mathsf{diverges} = \lambda x\colon \mathsf{OM}.\ b,\\ \quad \mathsf{requires} = \lambda x\colon \mathsf{OM}.\ qc \cdot x \wedge\\ \qquad \mathsf{variant} \cdot x = a_2,\\ \quad \mathsf{statement} = \mathsf{CATCH\text{-}CONTINUE} \cdot \ell \cdot s,\\ \quad \mathsf{ensures} = \lambda x\colon \mathsf{OM}.\\ \qquad \mathsf{invariant} \cdot x \wedge\\ \qquad \mathsf{variant} \cdot x < a_1,\\ \quad \mathsf{signals} = r,\\ \quad \mathsf{return} = R,\\ \quad \mathsf{break} = B,\\ \quad \mathsf{continue} = C\ )\end{array}}{\begin{array}{l}\mathsf{SB} \cdot (\ \mathsf{diverges} = \lambda x\colon \mathsf{OM}.\ b,\\ \quad \mathsf{requires} = \mathsf{invariant},\\ \quad \mathsf{statement} = \mathsf{WHILE} \cdot \ell \cdot c \cdot s,\\ \quad \mathsf{ensures} = q,\\ \quad \mathsf{signals} = r,\\ \quad \mathsf{return} = R,\\ \quad \mathsf{break} = B,\\ \quad \mathsf{continue} = C\ )\end{array}}$$

Fig. 6. Rule for total reasoning with while

The parameter $\ell\colon \mathsf{lift}[\mathsf{string}]$ in the goal statement $\mathsf{WHILE} \cdot \ell \cdot c \cdot s$ is $\mathsf{up}(\text{“}\texttt{lab}\text{”})$ if there is label `lab` immediately before the while statement in Java, and bot otherwise. If a statement `continue` or `continue lab` is executed within the loop body s, the resulting "continue" abnormality is caught by the wrapper $\mathsf{CATCH\text{-}CONTINUE} \cdot \ell \cdot s$, so that the next cycle can start normally. The LOOP tool puts a CATCH-BREAK function around every while statement, in order to catch any breaks within this statement[2]. The variant is a function $\mathsf{OM} \to A$ to some well-founded order A, which is required to decrease with every normally executed cycle[3]. Notice how an auxiliary predicate qc and values $a_1, a_2 \in A$ are used to pass on the effect of the condition to the statement—in the case where the condition evaluates to true. In this way the variant can decrease during execution of either the condition c or the statement s.

6 Example Verification in PVS

The rules from the previous section (plus some more rules) have all been formulated in PVS, and proven correct. This makes it possible to use these rules to prove that Java methods meet their JML specifications in PVS. The translations of these specifications are Boolean expressions of the form $\mathsf{SB} \cdot (\,\mathsf{diverges} = d, \cdots)$ or $\mathsf{EB} \cdot (\,\mathsf{diverges} = d, \cdots)$ involving suitable labeled tuples. These tuples can become very big during proofs, but the explicit labels keep them reasonably well-structured and manageable. The proof rules allow us to rewrite these labeled tuples into adapted tuples, following the structure of the Java code (of the body of the method whose correctness should be proved). This rewriting is continued until the statement or expression in the labeled tuple is sufficiently simple to proceed with the proof at a purely semantical level (like in the rule for RETURN in Subsection 5.3).

In this way the example JML specification of the pattern-search from Figure 1 has been proved for the given Java implementation. The latter involves `return` statements inside a `while` loop, leading to abrupt termination and a break out of the loop, both when it becomes clear that the pattern is present and that it is absent. This presents a non-trivial verification challenge, not only because of these `return` statements but also because of the non-trivial (in)variant involved, see [10]. The proof makes essential use of the rule for while (once) and for if-then-else (three times), and also for composition (several times), following the structure of the Java code.

The same example has been used in [10], where it was verified with the special Hoare logic (from [10]) with separate triples for the different termination modes in Java. It is re-used here to enable a comparison. Such a comparison is slightly

[2] The effect of these CATCH-BREAK and CATCH-CONTINUE functions can be incorporated into the while rule in Figure 6, by adapting the break and continue predicates in the assumptions, but this complicates this rule even further.

[3] Note that requiring the existence of the variant restricts the use of this rule to terminating while loops. Therefore, this "total" while rule only really make sense when the divergence clause is constantly false.

tricky because when the proof was re-done with the proof rules for JML, both the variant and invariant were already known. Also, no time had to be spent on formulating the required correctness property in PVS, because this could all be done (more conveniently) in JML. Taking this into account, the new rules still give a considerable speed-up of the proof. The verification is no longer a matter of days, but has become a matter of hours. The main reason is that the correctness formulas in the new logic for JML combine all termination modes in a single formula, and thus requires only one rule per language construct, with fewer assumptions.

7 Conclusion

In this paper JML method specification have been transformed into correctness formulas in an associated logic. These formulas extend standard Hoare triples (and those from [10]) by combining all possible termination modes for Java, naturally following the (coalgebraic) representation of statements and expressions. The correctness formulas capture all essential ingredients for an axiomatic semantics for Java. In combination with the underlying low-level, memory-based semantics of Java, these rules for JML provide an efficient, powerful and flexible setting for tool-assisted verification of Java programs with JML annotations.

Acknowledgements. Thanks are due to Joachim van den Berg and Marieke Huisman for discussing various aspects of the rules for JML.

References

1. JavaCard API 2.1. `http://java.sun.com/products/javacard/htmldoc/`.
2. P. America. Designing an object-oriented language with behavioural subtyping. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, number 489 in Lect. Notes Comp. Sci., pages 60–90. Springer, Berlin, 1990.
3. K.R. Apt. Ten years of Hoare's logic: A survey—part I. *ACM Trans. on Progr. Lang. and Systems*, 3(4):431–483, 1981.
4. J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert, C. Choppy, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques*, number 1827 in Lect. Notes Comp. Sci., pages 1–21. Springer, Berlin, 2000.
5. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. Techn. Rep. CSI-R0019, Comput. Sci. Inst., Univ. of Nijmegen. To appear at TACAS'01., 2000.
6. A. Bhorkar. A run-time assertion checker for Java using JML. Techn. Rep. 00-08, Dep. of Comp. Science, Iowa State Univ. (`http://www.cs.iastate.edu/~leavens/JML.html`), 2000.
7. F.S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *Foundations of Software Science and Computation Structures*, number 1578 in Lect. Notes Comp. Sci., pages 135–149. Springer, Berlin, 1999.

8. M. Huisman. *Reasoning about JAVA Programs in higher order logic with PVS and Isabelle*. PhD thesis, Univ. Nijmegen, 2001.
9. M. Huisman and B. Jacobs. Inheritance in higher order logic: Modeling and reasoning. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, number 1869 in Lect. Notes Comp. Sci., pages 301–319. Springer, Berlin, 2000.
10. M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in Lect. Notes Comp. Sci., pages 284–303. Springer, Berlin, 2000.
11. M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. Techn. Rep. CSI-R0007, Comput. Sci. Inst., Univ. of Nijmegen. To appear in *Software Tools for Technology Transfer*, 2001.
12. B. Jacobs. A formalisation of Java's exception mechanism. Techn. Rep. CSI-R0015, Comput. Sci. Inst., Univ. of Nijmegen. To appear at ESOP'01., 2000.
13. B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about classes in Java (preliminary report). In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 329–340. ACM Press, 1998.
14. G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov and B. Rumpe, editors, *Behavioral Specifications of Business and Systems*, pages 175–188. Kluwer, 1999.
15. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Techn. Rep. 98-06, Dep. of Comp. Sci., Iowa State Univ. (`http://www.cs.iastate.edu/~leavens/JML.html`), 1999.
16. K.R.M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Inst. of Techn., 1995.
17. K.R.M. Leino, J.B. Saxe, and R. Stata. Checking java programs via guarded commands. In B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs. Proceedings of the ECOOP'99 Workshop*. Techn. Rep. 251, Fernuniversität Hagen, 1999. Also as Technical Note 1999-002, Compaq Systems Research Center, Palo Alto.
18. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd rev. edition, 1997.
19. D. von Oheimb. Axiomatic semantics for Java^{light} in Isabelle/HOL. Technical Report CSE 00-009, Oregon Graduate Inst., 2000. TPHOLS 2000 Supplemental Proceedings.
20. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, number 1102 in Lect. Notes Comp. Sci., pages 411–414. Springer, Berlin, 1996.
21. L.C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in Lect. Notes Comp. Sci. Springer, Berlin, 1994.
22. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S.D. Swierstra, editor, *Programming Languages and Systems*, number 1576 in Lect. Notes Comp. Sci., pages 162–176. Springer, Berlin, 1999.
23. E. Poll. A coalgebraic semantics of subtyping. In H. Reichel, editor, *Coalgebraic Methods in Computer Science*, number 33 in Elect. Notes in Theor. Comp. Sci. Elsevier, Amsterdam, 2000.

24. E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Smart Card Research and Advanced Application*, pages 135–154. Kluwer Acad. Publ., 2000.
25. E. Poll, J. van den Berg, and B. Jacobs. Formal specification of the JavaCard API in JML: the APDU class. *Comp. Networks Mag.*, 2001. To appear.
26. Loop Project. `http://www.cs.kun.nl/~bart/LOOP/`.
27. Extended static checker ESC/Java. Compaq System Research Center. `http://www.research.digital.com/SRC/esc/Esc.html`.

A Hoare Calculus for Verifying Java Realizations of OCL-Constrained Design Models

Bernhard Reus[1], Martin Wirsing[2], and Rolf Hennicker[2]

[1] School of Cognitive and Computing Sciences, University of Sussex at Brighton
bernhard@cogs.susx.ac.uk, Fax +44 1273 671 320
[2] Institut für Informatik, Ludwig-Maximilians-Universität München
[hennicke|wirsing]@informatik.uni-muenchen.de

Abstract. The Object Constraint Language OCL offers a formal notation for constraining the modelling elements occurring in UML diagrams. In this paper we apply OCL for developing Java realizations of UML design models and introduce a new Hoare-Calculus for Java classes which uses OCL as assertion language. The Hoare rules are as usual for while programs, blocks and (possibly recursive) method calls. Update of instance variables is handled by an explicit substitution operator which also takes care of aliasing. For verifying a Java subsystem w.r.t. a design subsystem specified using OCL constraints we define an appropriate realization relation and illustrate our approach by an example.

1 Introduction

Program verification is a dream which has not yet been realized in practical software development. With UML [17] the possibilities for achieving this dream have improved: UML allows one to express semantic constraints using OCL and offers notations such as the "realizes" relation for expressing correctness relationships between different diagrams on different levels of abstraction. The object constraint language OCL [23] offers a formal notation to constrain the interpretation of modelling elements occurring in UML diagrams. OCL is systematically used for rigorous software development in the Catalysis Approach [11]. The OCL notation is particularly suited to constrain class diagrams since OCL expressions allow one to navigate along associations and to describe conditions for object attributes in invariants and pre- and post-conditions of the operations. The "realizes" relationship asserts that classes (written in a programming language) "realize" the requirements formulated in a more abstract class diagram with constraints. It allows the programmer to express the correctness of its implementations w.r.t. UML designs. However, to our knowledge, there is up to date no formal definition of the "realizes" relationship and also no possibility of verification.

The aim of this paper is to close this gap. We propose a formalization of the "realizes" relationship w.r.t. Java implementations. For this purpose we first define the syntactic and semantic requirements induced by a design model that

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 300–317, 2001.

is given by a UML class diagram and associated OCL constraints. The semantical requirements are presented by Hoare formulas which express pre- and post-conditions for the method implementations. We verify these requirements using the axioms and rules of a new Hoare calculus for Java-like sequential programs proposed in [20]. Even if in practice such proofs will not be done in full our approach provides a tool for verifying the critical and important parts of a realization relationship.

Our work has been influenced by several other calculi. A Hoare calculus for Java has been proposed by Poetzsch-Heffter and Müller [18,19]. We see as a main drawback that *loc.cit.* use an explicit representation of state in their calculus which is thus not suited for OCL. Similarly the calculi of Abadi and Leino [1,15] do not fit to Java and OCL. Our calculus avoids this problem and is directly tuned to OCL. In this sense we follow rather the ideas of Gries and De Boer who handle arrays and references by explicit substitution [10]. Other relevant work on Hoare-calculi includes a calculus of records [7] and treatment of recursion [22].

The JML approach [14] extends the Java language such that programs (including exceptions) can be annotated by specifications. Proof obligations are generated but proofs can only be performed after a translation into a voluminous semantic description of Java that does not make use of a Hoare-logic but is of denotational flavour instead.

Our calculus extends the usual rules for while programs with blocks by rules for update of instance variables – handled by an explicit substitution operator which also takes care of aliasing –, for creation of objects – using a special constant –, for recursive method specifications – taking care of inheritance –, and method calls.

2 The General Method

During the development of complex software systems various documents on different levels of abstraction are produced ranging from analysis models to concrete implementations (in terms of some programming language code). In this paper we focus on the (formal) relationship between system design and implementation.

2.1 The Design Model

Following the Unified Process (cf. [21]) a design model can be presented by a design subsystem. We assume that the subsystem contains classes, inheritance and association relations such that any association is directed and equipped with a role name and a multiplicity at the association end. As an essential ingredient of our approach the elements of a design model will be equipped with OCL constraints for specifying properties like invariants of classes and pre- and post-conditions for the operations.

In the following we consider as an example the design model for a (simple) account subsystem of a bank application shown in Fig. 1. For any checking account there is a history which stores the amounts of all deposit operations performed on the account. To describe precisely the desired effects of the operations in terms of pre- and post-conditions we use OCL-constraints which also include appropriate invariants for the specialized account classes (cf. Table 1).

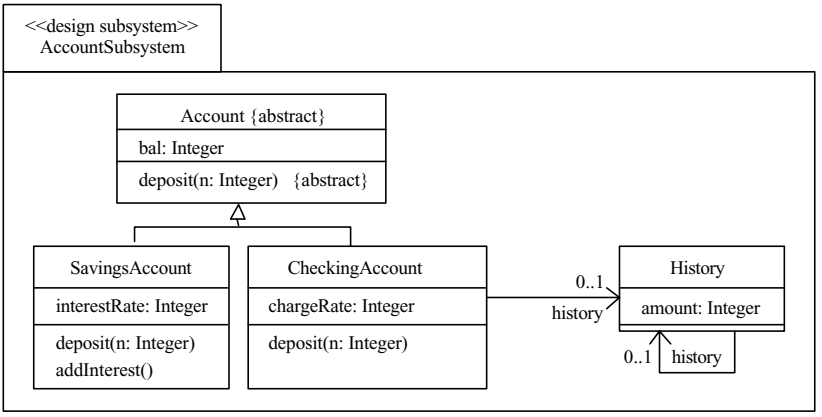


Fig. 1. Design Model for Accounts

2.2 The Implementation Model

An implementation model is given by an implementation subsystem (in the sense of [21]) which contains components that may be related by dependency relations. In our approach any component C.java will be a Java file containing the code of a Java class C. We assume that all attributes of Java classes are declared "private" or "private protected" to ensure encapsulation of object states. The code of the implementation model is shown in Table 1.

2.3 The Realization Relation

A realization relation connects a given design model and its corresponding implementation model as shown in Fig. 2. We say that the realization relation between "MyDesignSubsystem" and "MyJavaSubsystem" holds if the following syntactic and semantic requirements are satisfied.

Syntactic requirements: First the classes occurring in the design subsystem have to be mapped to components of the implementation model. This can be done by using trace dependencies as considered in [21]. We require that every class C of the design model is related by a trace dependency to a Java component C.java

Table 1. OCL-Constraints and implementation model for Account Subsystem

context Account::deposit(n:Integer)
pre: n ≥ 0
post: bal = bal@pre + n
	`abstract class Account`
	`{ private protected int bal;`
	`abstract void deposit(int n) {}`
	`}`

context SavingsAccount inv: bal ≥ 0 and interestRate ≥ 0	`class SavingsAccount extends Account` `{ private int interestRate;`

context SavingsAccount::
addInterest()
post: bal = bal@pre +
bal@pre * interestRate/100
	`public void deposit(int n)`
	`{ this.bal = this.bal + n;`
	`}`
	`public void addInterest()`
	`{ int interest = this.bal *`
	`this.interestRate/100;`
	`this.deposit(interest); }`

context CheckingAccount
inv: chargeRate ≥ 0
	`}`
	`class CheckingAccount extends Account`
	`{ private int chargeRate;`
	`private History history;`

context CheckingAccount::
deposit(n: Integer)
pre: n ≥ 0
post: history.oclIsNew and
history.amount = n and
history.history = history@pre
	`public void deposit(int n)`
	`{ this.bal = this.bal + n;`
	`History h = new History();`
	`h.amount = n;`
	`h.history = this.history;`
	`this.history = h; }`
	`}`
	`class History`
	`{ private int amount;`
	`private History history;`
	`}`

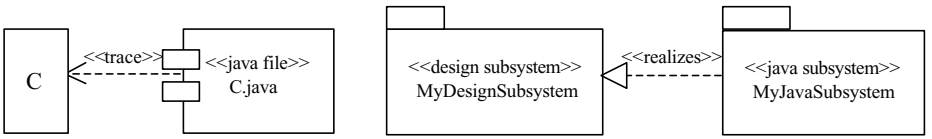


Fig. 2. Trace Dependency and Realization Relation

as depicted in Fig. 2. The trace dependency between C and C.java is supposed to hold if the following conditions are satisfied:

1. Each attribute of the design class C is also an attribute of the Java class C and for each role name at the end of a directed association the Java class contains a corresponding reference attribute with the same name. (Note that standard types may be slightly renamed according to the Java syntax

and that role names with multiplicity greater than one map to reference attributes of some container type.)
2. For each operation m specified in the design class C there is a method declaration in the Java class C and vice versa (up to an obvious syntactic modification of the signature). The operation m of the design model has the property { abstract } iff the method m is an abstract method.
3. The design class C is a (direct) subclass of a design class A iff the Java class C extends the Java class A.

These conditions guarantee (in particular) that the OCL expressions used as constraints for the design model can be interpreted in the implementation model which is necessary to define the semantical requirements. Moreover, note that the above conditions are satisfied by usual code generators for Java classes from UML class diagrams.

Semantic requirements: Let us first stress that the semantic requirements considered in the following are derived solely from the OCL constraints attached to the design model. This means that constraints imposed by the UML class diagram itself (like multiplicities or { query } properties of operations) and any kind of frame assumption will not be considered here if not explicitly expressed by an OCL constraint.

For the formulation of the semantic requirements we assume that the syntactic requirements from above are satisfied. Let us first discuss the role of invariants. According to [23] an invariant *INV-C* defined in the context of a class C means that *INV-C* evaluates to true for all instances of C at any moment of time. Since, by assumption, all attributes occurring in an implementation model are private or private protected the state of an object can only be modified by method invocations. Therefore the basic idea is to require that the invariant is preserved by any method invocation[1] for objects of C and that the invariant holds also for any object of C after its creation. These conditions, however, are not sufficient if there is a superclass A of C which has also an associated invariant, say *INV-A*. Then, in order to satisfy Liskov's substitution principle for subtypes [16], *INV-A* should be inherited by C. Hence, in general, we have to consider for any class C the conjunction of *INV-C* and all invariants *INV-A* associated to a superclass A of C. For any design class C, this conjunction will be denoted in the following by *INV-conj-C*. [2]

For dealing with object creation we transform any post-condition *POST* occurring in the design model into the expression *POST+* where any occurrence of an OCL expression "t.oclIsNew" with some term t of some type C is replaced by "t.oclIsNew and *INV-conj-C*[t/*this*]".

[1] For simplicity, we assume that all methods are public. Otherwise the approach could be easily extended to take into account UML visibility markers in the design model which then should be preserved by the trace dependency.

[2] Obviously, if *INV-C* is stronger than *INV-A* for any superclass A then *INV-conj-C* is equivalent to *INV-C*.

Having the above definitions in mind we require that for each design class C the following conditions are satisfied:

1. Pre- and post-conditions associated to operations of C are respected by corresponding method implementations. This means that for each operation m specified in the design class C with OCL-constraint

$$\text{context } C\text{:: m}(\mathrm{p}_1 : \mathrm{T}_1, \ldots, \mathrm{p}_n : \mathrm{T}_n)\text{pre} : \mathit{PRE}\ \text{post} : \mathit{POST}$$

   the given Java subsystem satisfies the Hoare formula

$$\{\mathit{PRE}\ \text{and}\ \mathit{INV\text{-}conj\text{-}C}\}\ C\texttt{::m(}\mathrm{p}_1 : \mathrm{T}_1, \ldots, \mathrm{p}_n : \mathrm{T}_n\texttt{)}\ \{\mathit{POST}+\}$$

   where C denotes the Java class with method m defined in the component C.java. The formal basis of this proof obligation will be provided in the next sections. In particular, according to Definitions 5 and 6, the satisfaction of the above Hoare formula means that any method body of m provided in C or in a subclass C' of C (which eventually overrides m) respects the given pre- and post-condition. Thus Liskov's substitution principle is satisfied.
   Note that it may also be the case that in the design model there is a subclass C' of the design class C which redefines m in the sense that it provides an additional OCL constraint with pre- and post-conditions PRE' and POST' for m. In this case the realization relation requires that both Hoare formulas

$$\{\mathit{PRE}\ \text{and}\ \mathit{INV\text{-}conj\text{-}C}\}\ C\texttt{::m(}\mathrm{p}_1 : \mathrm{T}_1, \ldots, \mathrm{p}_n : \mathrm{T}_n\texttt{)}\ \{\mathit{POST}+\}$$
$$\{\mathit{PRE}'\ \text{and}\ \mathit{INV\text{-}conj\text{-}C}'\}\ C'\texttt{::m(}\mathrm{p}_1 : \mathrm{T}_1, \ldots, \mathrm{p}_n : \mathrm{T}_n\texttt{)}\ \{\mathit{POST}'+\}$$

   must be satisfied by the Java subsystem. For instance, the pre- and post-conditions in our example lead to the proof obligations (1-3) of Table 2.
2. Invariants are preserved by method implementations. This means that for each operation m specified in the design class C or in a superclass of C the given Java subsystem satisfies the Hoare formula

$$\{\mathit{PRE}\ \text{and}\ \mathit{INV\text{-}conj\text{-}C}\}\ C\texttt{::m(}\mathrm{p}_1 : \mathrm{T}_1, \ldots, \mathrm{p}_n : \mathrm{T}_n\texttt{)}\ \{\mathit{INV\text{-}conj\text{-}C}\}$$

   where PRE denotes the pre-condition required for m (if any). For instance, considering the invariants of the account example we obtain the proof obligations (4-6) of Table 2.

3 OCL$^{\text{light}}$

OCL$^{\text{light}}$ is a representative kernel of OCL which should be easily extendible to full OCL. Yet, it deliberately differs from OCL in some minor syntactic points explained below.

Table 2. Proof obligations for the account example

{n ≥ 0} 1) `Account::deposit(n:Integer)` {bal = bal@pre + n}	{n ≥ 0 and bal ≥ 0 and interestRate ≥ 0} 4) `SavingsAccount::deposit(n:Integer)` {bal ≥ 0 and interestRate ≥ 0}
{n ≥ 0 and chargeRate ≥ 0} 2) `CheckingAccount::` `deposit(n:Integer)` {history.oclIsNew and history.amount = n and history.history = history@pre}	{bal ≥ 0 and interestRate ≥ 0} 5) `SavingsAccount::addInterest()` {bal ≥ 0 and interestRate ≥ 0}
{true} 3) `SavingsAccount::addInterest()` {bal = bal@pre+ bal@pre ∗ interestRate/100}	{n ≥ 0 and chargeRate ≥ 0} 6) `CheckingAccount::deposit(n:Integer)` {chargeRate ≥ 0}

3.1 Syntax

OCL$^{\text{light}}$ admits the use of so-called "logical variables" for eliminating expressions of the form "t@pre" from post-conditions. Such variables cannot be altered by any program. All other variables are simply referred to as "program variables". By contrast to Table 2, we stipulate that all instance variables are fully qualified, i.e. we write "this.bal" instead of just "bal".

Note that formal parameters of methods are assumed to appear as *logical variables* in assertions since they are not allowed to change (call-by-value). Moreover, we use "this" and "null" although the former is called "self" in OCL and the latter is expressed in OCL by the use of the formula "isEmpty", i.e. instead of "t→isEmpty" write "t = null". The OCL-term-syntax is extended by an operation "new(C)". It should not be used in OCL- specifications, but it may pop up in assertions during the verification process to cope with object creation (cf. Section 5.3). that is sound w.r.t. the above given interpretation function.

General OCL$^{\text{light}}$-terms may additionally be built from

$$\begin{array}{rll} t ::= & \langle Var \rangle.\text{a@pre} & \text{field variables in previous state} \\ \mid & \langle Var \rangle.\text{a.oclIsNew} & \text{test for new field variable} \end{array}$$

where $\langle Var \rangle$ must not be a logical variable.

OCL$^{\text{light}}$-*formulas* are expressions of type bool subsuming equality, forall, exists, and includes-expressions.

Notation: Usually we use capital letters (X, Y, Z) for logical variables and small ones for program variables. An exception from the rule are the formal parameters of methods which are uniquely identified by syntax and thus can remain lowercase although regarded logical.

3.2 Semantics of OCL$^{\text{light}}$-Terms

There is an interpretation function, $[\![\_]\!]_{\_,\_,\_,\_}$, taking a *pure* OCL$^{\text{light}}$-term, a store (containing the objects), a (runtime-) stack (containing actual parameters of methods and local variables), two environments – one for logical variables and one for query names –, and yields an element in a semantic domain. The definition of $[\![e]\!]_{\mu,\sigma,\beta,\rho}$ is by induction on e. It is rather straightforward and thus omitted (it can be found in [20]). However, query calls were not considered in *loc.cit.*, therefore we have introduced an environment for queries and the semantics of a call for query q is as follows:

$$[\![e_0.\mathrm{q}(\boldsymbol{e})]\!]_{\mu,\sigma,\beta,\rho} = \rho(\mathrm{q})([\![e_0]\!]_{\mu,\sigma,\beta,\rho}, [\![\boldsymbol{e}]\!]_{\mu,\sigma,\beta,\rho}, \mu)$$

where a query environment ρ maps a query name to a function $\rho(q)$, taking as input an object reference (the actual value of *this*), some arguments of a type determined by the argument types of the query and a store (which is needed to obtain the field values of *this*). Moreover, $[\![\boldsymbol{e}]\!]$ is the canonical generalisation of $[\![\_]\!]$ to a list of terms. In the following we will analogously use extensions of β and σ to lists of variables.

The axiomatization of the OCL$^{\text{light}}$-logic contains the usual axioms for natural numbers, typed finite set theory, and booleans. The "forall" and "exists" quantifiers are always bounded by a set. The two "non-standard" operations are "new(C)" representing a free object reference, and "allInstances" referring to all actually existing and valid objects of a certain class type. They are axiomatized as follows:

$$\mathrm{not}(\mathrm{new}(C) = \mathrm{null})$$
$$C.\mathrm{allInstances}{\rightarrow}\mathrm{includes}(t) \text{ iff } \mathrm{not}(t = \mathrm{null}) \text{ and } \mathrm{not}(t = \mathrm{new}(C))$$

where t is of type C, "iff" means "if, and only if" obtained from "implies". Since $\mathrm{not}(C.\mathrm{allInstances} \rightarrow \mathrm{includes}(t) = C.\mathrm{allInstances} \rightarrow \mathrm{includes}(t'))$ implies $\mathrm{not}(t = t')$ one can derive e.g. $C.\mathrm{allInstances} \rightarrow \mathrm{forall}(Y|\mathrm{not}(\mathrm{new}(C) = Y))$.

The queries need a bit of work too. If q is a query with precondition P and postcondition Q the following axiom is supposed to hold:

$$P[e_0/\mathrm{this}, \boldsymbol{e}/\boldsymbol{p}] \text{ implies } Q[e_0/\mathrm{this}, \boldsymbol{e}/\boldsymbol{p}, e_0.q(\boldsymbol{e})/\mathrm{result}]$$

This axiom is only sound, of course, if $\rho(\mathrm{q})$ obeys the specification given as pre- and post-condition which will be generally assumed in the following, i.e. we require for any ρ that for any OCL-query-specification in a set of class declarations D, context $\mathrm{q}(\boldsymbol{p} : \boldsymbol{\tau}) : \tau_r$ pre : P post : Q, it holds that

$$\forall \mu, \sigma, \beta.\, [\![P_q]\!]_{\mu,\sigma,\beta,\rho} = \mathit{true} \text{ implies } [\![Q_q]\!]_{\mu,\sigma[\mathit{result}\rightarrow\rho(q)(\sigma(\mathit{this}),\beta(\boldsymbol{p}),\mu)],\beta,\rho} = \mathit{true}$$

abbreviated to $\rho \Vdash \mathrm{Queries}(D)$.

Note that formal parameters are treated as logical variables. This is justified by the assumption that we only deal with call-by-value parameter-passing.

Theorem 1. (Soundness) *There is an axiomatization of the OCL$^{\text{light}}$-logic with pure terms, $\vdash_l$, such that for any pure OCL$^{\text{light}}$-formula Q it holds that*

$$\vdash_l Q \;\Rightarrow\; \forall \mu, \sigma, \beta, \rho.\ \rho \Vdash \text{Queries}(D) \text{ implies } [\![Q]\!]_{\mu,\sigma,\beta,\rho} = \mathit{true}$$

For general OCL$^{\text{light}}$-terms the interpretation function has an additional parameter representing the *old* store, i.e. the interpretation function is written $[\![e]\!]^{\mu_p}_{\mu,\sigma,\beta,\rho}$ where μ_p denotes the old store, whereas μ stands for the actual one.

Definition 1. *The interpretation function for* general *OCL$^{\text{light}}$-terms is defined inductively such that $[\![t@pre]\!]^{\mu_p}_{\mu,\sigma,\beta,\rho} = [\![t]\!]_{\mu_p,\sigma,\beta,\rho}$ and $[\![t.oclIsNew]\!]^{\mu_p}_{\mu,\sigma,\beta,\rho} = ([\![t]\!]_{\mu,\sigma,\beta,\rho} \not\in \mu_p)$ where $x \not\in \mu_p$ is true iff x is not referring to an existing object in μ_p. All other cases follow literally the interpretation for the pure case.*

3.3 Transformation of OCL$^{\text{light}}$-Formulas

Postconditions may contain expressions of the form "t.a@pre" and "t.oclIsNew" which are forbidden in preconditions. This is impractical in proofs of Hoare-formulas where the postcondition of one statement may appear as precondition of another statement. Therefore we introduce logical variables for encoding the effects of "@pre" and "oclIsNew".

Definition 2. *For a pair of general OCL$^{\text{light}}$-formulas (P, Q) we define the syntactic transformation $(P, Q)^* = (P^*, Q^*)$ as follows:*

$$P^* = (P \text{ and } t_i = X_i \text{ and } C_j.\text{allInstances} = A_j)$$
$$Q^* = Q[X_i/t_i@\text{pre}][\text{not}(e_j = \text{null}) \text{ and not}(A_j{\rightarrow}\text{includes}(e_j))/e_j.\text{oclIsNew}]$$

where $\{t_i@\text{pre} \mid i \in I\}$ contain all occurrences of "@pre"-variables in Q and $\{e_j.\text{oclIsNew} \mid j \in J\}$ contain all occurrences of "oclIsNew" in Q. The C_j are the class types of the e_j. All X_i and A_j are new logical variables not *occurring in P or Q.*

Example 1. We can transform the pre- and postcondition of the `deposit` operation in `Account` to the following Hoare-formula:

{ this.bal = M and n $\geq$ }`Account::deposit(Integer n)`{ this.bal = M + n }

The "oclIsNew" part of the pre-/postcondition of deposit in CheckingAccount is transformed as follows (written vertically):

{ History.allInstances = H and n $\geq$ 0 }
`CheckingAccount::deposit(Integer n)`
{ not(this.history = null) and not(H→includes(this.history)) and ... }

3.4 Correctness of the Transformation

Proposition 1. *Let P, Q be general OCL$^{\text{light}}$-formulas and $(P, Q)^* = (P^*, Q^*)$. Then for all $\mu, \mu_p, \sigma, \beta$ and ρ we have*

$$[\![P]\!]_{\mu_p,\sigma,\beta,\rho} \Rightarrow [\![Q]\!]^{\mu_p}_{\mu,\sigma,\beta,\rho} \text{ iff } [\![P^*]\!]_{\mu_p,\sigma,\beta^*,\rho} \Rightarrow [\![Q^*]\!]_{\mu,\sigma,\beta^*,\rho}$$

$$\textit{where} \qquad \beta^*(Z) = \begin{cases} [\![t_i]\!]_{\mu_p,\sigma,\beta,\rho} & \text{if} \quad Z \equiv X_i, i \in I \\ [\![C_j.\text{allInstances}]\!]_{\mu_p,\sigma,\beta,\rho} & \text{if} \quad Z \equiv A_j, j \in J \\ \beta(Z) & \text{otherwise} \end{cases}$$

and $(t_i)_{i\in I}$ and $(C_j)_{j\in J}$ are as in Def. 2.

4 Java$^{\text{light}}$

The object-oriented programming language of discourse is supposed to be a subset of sequential Java with methods and constructors without exceptions.

There are some restrictions, however, on the syntax that deserve explanation. First, we do not allow arbitrary assignments *Exp*`.a =` *Exp* as we will only be able to define substitution for instance (field) variables $x.a$ where x is a local variable or a formal parameter (or `this`). This is, however, no real restriction as for an assignment e`.a =`e' one can also write x`=`e`;` x`.a =`e'. This sort of a decomposition of compound expressions is well known from compiler construction. Second, we distinguish a subset of expressions without side-effects (*Exp*) and with possible side-effects (*Sexp*). The first forms a proper subset of OCL$^{\text{light}}$-expressions and can thus be substituted for (instance) variables. This is why all the arguments of a method call must be side-effect free. The restricted syntax for expressions is still sufficient since, again, one can decompose any expression using auxiliary variables.

In general, dealing with partial correctness only, we shall only consider verification of programs that are syntax and type correct. For technical simplicity two minor simplifications of the Java type-system are in use. We ignore shadowing of field variables and method overloading (by different number and types of argument variables).

Semantics. For the purpose of this paper it is sufficient to treat the operational semantics of Java$^{\text{light}}$ abstractly.

Definition 3. *An operational semantics for Java$^{\text{light}}$ is a family of partial functions*

$$(\mathcal{T}^C)_{C\in Classname} : JavaL \times Store \times Stack \rightharpoonup Store \times Stack$$

that is defined – assuming that this has actual type C – only if execution of the Java-program terminates successfully. Moreover, the result has to be in accordance with the requirements of the Java Specification [12]. The restriction $\mathcal{T}^C_k$ yields the same result as $\mathcal{T}^C$ if the evaluation depth (the call-stack-depth) of the computation is less than k; otherwise it is undefined.

This restriction is necessary to give a sound interpretation to specifications of recursive methods (see also [22,20]).

A possible operational semantics that fits can be found in [8,9,2]).

5 Hoare Calculus

In this section we present a Hoare calculus for Java$^{\text{light}}$ with assertions written in *pure* OCL$^{\text{light}}$. This calculus extends the well-known Hoare calculi one can find in any textbook (see e.g. [4] or the original text [13]) by a few rules covering assignment to instance variables, object creation (see also [6,7]), method call, and method specification (inheritance).

5.1 Syntax

Because object-oriented programs are structured by means of classes which in turn break down to fields and methods, we introduce two different Hoare-like judgements, where the one for methods is considered as a special abbreviation:

Definition 4. *We first distinguish between two types of Hoare-triples, those* for statements $\{P\}\ S\ \{Q\}$ *and those for* methods *(also called method specifications)* $\{P\}\ C :: m(\boldsymbol{p} : \boldsymbol{\tau})\ \{Q\}$ *where S is a Java$^{\text{light}}$-statement, C is a class type, P and Q are pure OCL$^{\text{light}}$ formulas. For method specifications, all program variables appearing in Q must be "this" or "result". Recall that the formal parameters $\boldsymbol{p}$ are assumed to appear as* logical *variables since we assume a call-by-value parameter mechanism. The judgments of the Hoare calculus are then as follows:*

1. Derivable Statement Triples
   $\Gamma \vdash^{C}_{D} \{P\}\ S\ \{Q\}$ *where Γ denotes a context being empty or consisting of one method specification, D is the whole set of declarations, i.e. the complete Java-package of discourse, and C is the assumed class type of* `this` *(which may not be uniquely derivable from the statement S alone).*
2. Derivable Method Triples
   $\vdash_{D} \{P\}\ C :: m(\boldsymbol{p} : \boldsymbol{\tau})\ \{Q\}$ *where C and D are as above.*

The context for Hoare triples is necessary for the treatment of recursive method specifications. For *mutual* recursive methods the context must be generalized to sets of method triples.

We omit the indices D and C if they can be derived from the context.

5.2 Semantics

The following definition of validity of triples holds for *general* OCL$^{\text{light}}$-assertions P and Q.

Definition 5. *Let $\mathcal{T}$ denote a semantic function for Java$^{\text{light}}$. Then Hoare-triples are said to hold relatively to evaluation depth smaller than k if the following holds:*

1. Statement Triples *(partial correctness of statements)*
$\models_k^{D,C} \{P\}\ S\ \{Q\}$, *if for any* μ, σ, β *we have*

$$[\![P]\!]_{\mu,\sigma,\beta,\rho} = \mathit{true} \wedge \mathcal{T}_k^C(S,\mu,\sigma) = (\mu',\sigma') \Rightarrow [\![Q]\!]^{\mu}_{\sigma',\mu',\beta,\rho} = \mathit{true}$$

where ρ *is defined as follows for any query of* D *defined in class* C_q*:*

$$\rho(q)(o,\boldsymbol{a},\mu) = (\mathcal{T}_k^{C_q}(\mathrm{body}(C_q,q,D),\mu,\emptyset[\mathit{this} \mapsto o][\boldsymbol{p} \mapsto \boldsymbol{a}])_1(\mathit{result})$$

2. Method Triples *(partial correctness of methods)*
$\models_k^D \{P\}\ C\texttt{::}\boldsymbol{m}(\boldsymbol{p}:\boldsymbol{\tau})\ \{Q\}$ *if* $\forall C' \leq C.\ \models_k^{D,C'} \{P\}\ \mathrm{body}(C',\boldsymbol{m},D)\ \{Q\}$ *where* $\mathrm{body}(C',\boldsymbol{m},D)$ *is the body of* $\boldsymbol{m}$ *defined in class* C' *of program package* D*. If* C' *just inherits* $\boldsymbol{m}$ *from some superclass* C'' *then* $\mathrm{body}(C',\boldsymbol{m},D) = \mathrm{body}(C'',\boldsymbol{m},D)$.

Note that it is not clear *a priori* that $\rho \Vdash \mathrm{Queries(D)}$, but it will follow from the proof of $\vdash_D \{P\}\ C\texttt{::q(...)}\ \{Q\}$ for each query q with pre-condition P and post-condition Q.

Definition 6. *A triple* T *is valid in a context* Γ*, i.e.* $\Gamma \models^{D,C} T$*, iff*

$$\forall k \in \mathbb{N}.\ \models_k^D \Gamma \Rightarrow \models_{k+1}^{D,C} T$$

5.3 Inductive Definition of the Hoare Calculus

In this section we present the rules (i.e. the calculus) for deriving *correct* specifications for Java$^{\mathrm{light}}$ programs in a purely syntactic way. The rules and axioms below define inductively a relation $\vdash_C^D$, i.e. the derivable statement specifications.

To this end we may make use of the axioms and rules for the OCL$^{\mathrm{light}}$-language (i.e. $\vdash_l$, cf. Theorem 1) and of the "classical" rules of the Hoare calculus for While-languages which are not repeated here (cf. [13,3]).

In the following we present the rules that deal with object-oriented features.

Field Assignment

$$\{P[e/x.a]\}\ x\texttt{.a=}e\ \{P\} \qquad e \in \mathit{Exp} \qquad \text{(Field variable assignment)}$$

where $t[e/x.a]$ is the substitution for field variables defined inductively as follows:

Definition 7. *Define* $e'[e/x.a]$ *by structural induction on* e'*, the only interesting non-trivial case being (in other cases just push substitution through):*

$$(t.b)[e/x.a] \triangleq \begin{cases} t[e/x.a].b & \mathit{if}\ b \not= a \\ \mathit{if}\ (t[e/x.a] = x)\ \mathit{then}\, e\ \mathit{else}\ t[e/x.a].b & \mathit{otherwise} \end{cases}$$

Example 2. The following Hoare-triple is an instance of the field variable assignment axiom for the body of the `deposit` operation in `SavingsAccount`:

{ (if (this = this) then (this.bal + n) else this.bal) = M + n}
`this.bal = this.bal + n`
{ this.bal = M + n }

which by the Weakening Rules reduces to

$$\{\text{this.bal} + \text{n} = \text{M} + \text{n}\}\ \texttt{this.bal = this.bal + n}\ \{\text{this.bal} = \text{M} + \text{n}\}$$

Again by weakening we obtain the correctness of the body of the method `deposit` of class `SavingsAccount` w.r.t. the transformed OCL$^{\text{light}}$-pre/post-condition of `deposit` given in the superclass `Account` (cf. (1) of Table 2).

Object creation. Let $Q[\delta_{C.a}/x.\boldsymbol{a}]$ abbreviate the simultaneous substitution of all field variables $x.a_i$ occurring in Q by a default value of appropriate type. This default value has to be the one that Java$^{\text{light}}$ uses for initialisation (e.g. 0 for integers and null for class types).

$\{Q[\delta_{C.a}/x.\boldsymbol{a}][\text{new}(C)/x, C.\text{allInstances}{\rightarrow}\text{including}(\text{new}(C))/C.\text{allInstances}]\}$
`x=new` C`()` (new)
$\{Q\}$

where Q does not contain any query calls nor new(C).

Recall that "new(C)" and query calls can be eliminated using the consequence rule of standard Hoare calculus and the axioms mentioned in Section 3.2.

Example 3. The correctness of `deposit` in `CheckingAccount` involves proving the following Hoare-formula (*):

{ H = History.allInstances }
`History h = new History()`
{ not($H{\rightarrow}$includes(h)) and not(h = null) }

Using the axiom for object creation the derived precondition is

$$(**) \qquad \text{not}(H{\rightarrow}\text{includes}(\text{new}(\text{History}))) \text{ and } \text{not}(\text{new}(\text{History}) = \text{null})$$

Because of the axioms for "new(History)" and "History.allInstances" the precondition of (*) implies (**). Thus by the weakening rule, the Hoare-formula (*) is proven.

return-statement. Returning a value means assigning it to variable *result*.

$$\{Q[e/\text{result}]\}\ \texttt{return}\ e\ \{Q\} \qquad \text{(return)}$$

Method specifications. The partial correctness of a method specification for `m` in class C can be derived from the partial correctness of all bodies of `m` in C and

in any subclass of C where for dealing with recursion the partial correctness of the method specification can be assumed.

$$\frac{\forall C' \leq C.\, \{P\}\ C\texttt{::m(}\boldsymbol{p}\texttt{:}\boldsymbol{\tau}\texttt{)}\ \{Q\} \vdash^{D}_{C'}\ \{P\}\ \mathrm{body}(C', \texttt{m}, D)\ \{Q\}}{\{P\}\ C :: \texttt{m}(\boldsymbol{p} : \boldsymbol{\tau})\ \{Q\}} \quad \text{(MethodSpec)}$$

Example 4. By proving the correctness of the method bodies of `deposit` in `SavingsAccount` (cf. Example 2) and `CheckingAccount` i.e.

$$\begin{array}{ll} \vdash^{\text{AccountJavaSubsystem}}_{\text{SavingsAccount}} & \{\ \text{this.bal} = \text{M and n} \geq 0\ \} \\ & \text{body(SavingsAccount,}\texttt{deposit}\text{,AccountJavaSubsystem)} \\ & \{\ \text{this.bal} = \text{M} + \text{n}\ \} \qquad \text{and} \\ \vdash^{\text{AccountJavaSubsystem}}_{\text{CheckingAccount}} & \{\ \text{this.bal} = \text{M and n} \geq 0\ \} \\ & \text{body(CheckingAccount,}\texttt{deposit}\text{,AccountJavaSubsystem)} \\ & \{\ \text{this.bal} = \text{M} + \text{n}\ \} \end{array}$$

we conclude the correctness of `deposit` w.r.t. its transformed OCL-constraint by rule (MethodSpec).

$$\begin{array}{ll} \vdash_{\text{AccountJavaSubsystem}} & \{\ \text{this.bal} = \text{M and n} \geq 0\ \} \\ & \texttt{Account::deposit(n : Integer)} \\ & \{\ \text{this.bal} = \text{M} + \text{n}\ \} \end{array}$$

Method Call. The rules for the method call must take into consideration the method dispatch of the programming language. This is ensured by using the method specification in the premise.

$$\frac{\{P\}\ \mathit{type}(e_0)\texttt{::m(}\boldsymbol{p}\texttt{:}\boldsymbol{\tau}\texttt{)}\ \{Q\} \qquad \vdash_l Q[e_0/\text{this}]\ \text{implies}\ R[\text{result}/x]}{\{P[e_0/\text{this}]\ \text{and}\ \boldsymbol{p} = \boldsymbol{e}\}\ x\texttt{=}e_0\texttt{.m(}\boldsymbol{e}\texttt{)}\ \{R\}} \quad \text{(Call)}$$

Note that one cannot simplify the rule by dropping the implication in the hypothesis and changing the postcondition in the conclusion to $Q[e_0/\,\mathit{this}, x/\mathit{result}]$ since this would blur the distinction between x before and after execution of the method call and thus lead to an unsound rule. For the very same reason the arguments $\boldsymbol{e}$ cannot be substituted into Q.

Logical variables can be replaced by special side-effect free expressions.

$$\frac{\{P\}\ x\texttt{=}e_0\texttt{.m(}\boldsymbol{e}\texttt{)}\ \{Q\}}{\{P[\boldsymbol{e'}/\boldsymbol{Z}]\}\ x\texttt{=}e_0\texttt{.m(}\boldsymbol{e}\texttt{)}\ \{Q[\boldsymbol{e'}/\boldsymbol{Z}]\}} \quad \text{if } \boldsymbol{e'} \in \mathit{Exp},\ x \not\in LV(\boldsymbol{e'}), IV(\boldsymbol{e'}) = \emptyset$$

(Call Invariance)

where $LV(\boldsymbol{e'})$ and $IV(\boldsymbol{e'})$ denote the local variables and the instance variables occurring in vector $\boldsymbol{e'}$, respectively, and $\boldsymbol{Z}$ is a vector of logical variables (thus not occurring in any program). The variable conditions ensure that $\boldsymbol{e'}$ is not changed by the method invocation.

For method calls with return type `void` there is an analogous rule.

$$\frac{\{P\}\ \mathit{type}(e_0)\texttt{::m(}\boldsymbol{p}\texttt{:}\boldsymbol{\tau}\texttt{)}\ \{Q\}}{\{P[e_0/\text{this}]\,\text{and}\,\boldsymbol{p} = \boldsymbol{e}\}\ e_0\texttt{.m(}\boldsymbol{e}\texttt{)}\ \{Q[e_0/\text{this}]\}} \quad \text{(CallVoid)}$$

We omit the analogous invariance rule for methods with return type.

Example 5. In the following we prove a property of `deposit` which is used in the proof of the constraint for `addInterest`:

$$\dfrac{\dfrac{\{\text{this.bal} = \text{M and n} \geq 0\}\ \texttt{SavingsAccount::deposit(n:Integer)}\ \{\text{this.bal} = \text{M+n}\}}{\{\text{this.bal} = \text{M and n} \geq 0 \text{ and n=interest}\}\ \texttt{this.deposit(interest)}\ \{\text{this.bal=M+n}\}}(\text{MethodCall})}{\{Q\}\ \texttt{this.deposit(interest)}\ \{\text{this.bal =M+M*I/100}\}}(\text{CallInvariance})$$

where $Q \equiv$ "this.bal = M and M*I/100 $\geq$ 0 and M*I/100=interest" and "I" is a logical variable denoting the value of "this.interestRate". Note that for proving "M*I/100 $\geq$ 0" we need the invariant of `SavingsAccount` asserting "this.bal $\geq$ 0 and this.interestRate $\geq$ 0".

5.4 Correctness

Theorem 2. *The presented Hoare calculus for pure OCL$^{\text{light}}$-formulas and Java$^{\text{light}}$ programs is sound w.r.t. the operational semantics of Java$^{\text{light}}$ given in [9], i.e.*

$$\Gamma \vdash_D^C \{P\}\ S\ \{Q\} \Rightarrow \Gamma \models^{D,C} \{P\}\ S\ \{Q\}$$

Proof. [20]

Corollary 1. *For general OCL$^{\text{light}}$-formulas P and Q we therefore get*

$$\Gamma \vdash_D^C \{P^*\}\ S\ \{Q^*\} \Rightarrow\ \Gamma \models^{D,C} \{P\}\ S\ \{Q\}$$

Proof. The proof is a consequence of the theorem above and Proposition 1.

Currently we are investigating the completeness of the Hoare calculus. It appears that we need some additional (admissible) rules such as conjunction introduction and the introduction of existential quantifiers, see e.g. [4].

6 Verifying the Realization Relation

In this section we sketch the proof of the correctness of the realization relation of the AccountSubsystem (see Fig. 3). According to the definition in Section 2.3 we have to show the trace dependencies, the satisfaction of the pre-/postcondition constraints and the preservation of the OCL-invariants.

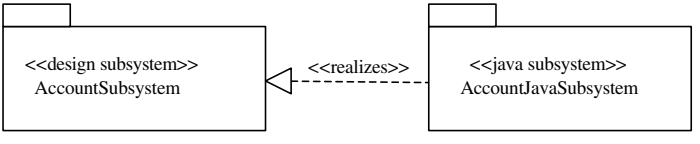


Fig. 3. Realization relation of the AccountSubsystem

Trace dependencies. The trace dependencies are obviously satisfied: for each class of AccountSubsystem there exists a corresponding Java class in AccountJavaSubsystem so that the attributes, methods and inheritance relations are preserved.

Satisfaction of pre-/postconditions. The proof obligations (1-3) of Table 2 shown in Section 2.3 have to be verified. For this purpose, according to Corollary 1, it is sufficient to consider their transformations which can be proved as sketched in Example 4 (for (1)), Example 3 (for (2)), and Example 5 (for (3)).

Preservation of invariants. The AccountSubsystem contains invariants for SavingsAccount and CheckingAccount. It is easy to prove the associated conditions (4-6) shown in Section 2.3.

7 Concluding Remarks

We have presented a new formal approach for verifying the realization of UML design models by Java subsystems and a new Hoare calculus for a sequential subset of Java and a subset of OCL as assertion language. This is a first step towards the goal of providing a basis for formal software development with UML. But one can see several necessary extensions of our approach, for the UML part as well as for the Hoare calculus. In this paper we have restricted the design models to classes and their relationships. In the following we plan to consider also interfaces. Here, our approach of [5] where we propose a constraint language for interfaces may provide a good basis for the extension. Another important question concerns the composability of subsystems: Under which conditions is the correctness of the realizes relationship preserved if two subsystems with correct realizations are composed? Concerning the Hoare calculus it should be easy to extend semantics, calculus, and soundness proof to the full OCL-language (with bags, sequences and many operations on them). In order to analyse the practicability of our calculus we also need to carry out further case studies. Those examples might then propose additional admissible or derived proof-rules for the Hoare calculus in order to support the verification process, i.e. to simplify the reasoning.

Acknowledgement. Thanks to Hubert Baumeister for useful discussions on the realization relation and Alexander Knapp for helping with the conversion of the diagrams.

References

1. M. Abadi and K.R.M. Leino. A logic of object-oriented programs. In Michel Bidoit and Max Dauchet, editors, *Theory and Practice of Software Development: Proceedings / TAPSOFT '97, 7th International Joint Conference CAAP/FASE*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer-Verlag, 1997.

2. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes Comp. Sci.* Springer, Berlin, 1999.
3. K.R. Apt. Ten Years of Hoare's Logic: A Survey – Part I. *TOPLAS*, 3(4):431–483, 1981.
4. K.R. Apt and E.R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer, 1991.
5. M. Bidoit, R. Hennicker, F. Tort, and M. Wirsing. Correct realizations of interface constraints with OCL. In *UML'99, The Unified Modeling Language - Beyond the Standard, Fort Collins, USA*, volume 1723 of *Lecture Notes in Computer Science.* Springer-Verlag, 1999.
6. R. Bornat. Pointer aliasing in Hoare logic. In *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 102–126. Springer-Verlag, 2000.
7. C. Calcagno, S. Ishtiaq, and P.W. O'Hearn. Semantic analysis of pointer aliasing, allocation and disposal in Hoare logic. In *Principles and Practice of Declarative Programming.* ACM Press, 2000.
8. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. From Sequential to Multi-Threaded Java: An Event-Based Operational Semantics. In M. Johnson, editor, *Proc. 6*th *Int. Conf. Algebraic Methodology and Software Technology*, volume 1349 of *Lect. Notes Comp. Sci.*, pages 75–90, Berlin, 1997. Springer.
9. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An Event-Based Structural Operational Semantics of Multi-Threaded Java. In Alves-Foss [2], pages 157–200.
10. F.S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *Foundations of Software Science and Computations Structures*, volume 1578 of *Lecture Notes in Computer Science.* Springer-Verlag, 1999.
11. D. D'Souza and A.C. Wills. *Objects, components and frameworks with UML: the Catalysis approach.* Addison–Wesley, Reading, Mass., etc., 1998.
12. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison–Wesley, Reading, Mass., 1996.
13. C.A.R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12:576–583, 1969.
14. Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *Object Oriented Programming: Systems, Languages and Applications (OOPSLA)*, volume 35, pages 208–228. ACM SIGPLAN Notices, 2000.
15. K. Rustan M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. Technical Report KRML 65-0, SRC, 1996.
16. B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.
17. Object Management Group. Unified Modeling Language – Object Constraint Language Specification. Technical report, available at http://www-4.ibm.com/software/ad/standards/ad970808_UML11_OCL.pdf, 1998.
18. A. Poetzsch-Heffter and P. Müller. A logic for the verification of object-oriented programs. In R. Berghammer and F. Simon, editors, *Programming Languages and Fundamentals of Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1997.
19. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science. Springer-Verlag, 1999.
20. B. Reus and M. Wirsing. A Hoare-Logic for Object-oriented Programs. Technical report, LMU München, 2000.

21. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison–Wesely, Reading, Mass., etc., 1998.
22. D. von Oheimb. Hoare logic for mutual recursion and local variables. In V. Raman C. Pandu Rangan and R. Ramanujam, editors, *Found. of Software Techn. and Theoret. Comp. Sci.*, volume 1738 of *LNCS*, pages 168–180. Springer, 1999.
23. J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison–Wesley, Reading, Mass., etc., 1999.

A Formal Object-Oriented Analysis for Software Reliability: Design for Verification

Natasha Sharygina[1], James C. Browne[2], and Robert P. Kurshan[1]

[1] Bell Laboratories, 600 Mountain Ave.,
Murray Hill, NJ, USA 07974
{natali,k}@research.bell-labs.com
[2] The University of Texas at Austin, Computer Science Department,
Austin, TX, USA 78712
browne@cs.utexas.edu

Abstract. This paper presents the OOA design step in a methodology which integrates automata-based model checking into a commercially supported OO software development process. We define and illustrate a set of design rules for OOA models with executable semantics, which lead to automata models with tractable state spaces. The design rules yield OOA models with functionally structured designs similar to those of hardware systems. These structures support model-checking through techniques known to be feasible for hardware. The formal OOA methodology, including the design rules, was applied to the design of NASA robot control software. Serious logical design errors that had eluded prior testing, were discovered in the course of model-checking.

1 Introduction

Problem Statement. Software systems used for control of modern devices are typically both complex and concurrent. Object-Oriented (OO) development methods are increasingly employed to cope with the complexity of these software systems. OO development systems still largely depend on conventional testing to validate correctness of system behaviors, however. This is simply not adequate to attain the needed reliability, for complex systems, on account of the intrinsic incompleteness of conventional testing.

Formal verification of system behavior through model checking [4], on the other hand, formally verifies that a given system satisfies a desired behavioral property through exhaustive search of ALL states reachable by the system. Model checking has been widely and successfully applied to verification of hardware systems. Application of model checking to software systems, has, in contrast, been much less successful. To apply model checking to software systems the software systems must be translated from programming or specification languages to representations to which model checking can be applied. The resulting representation for model checking must have a tractable state space if model checking is to be successful. Translation of software systems designed by conventional development processes and even by OO development processes to representations to which model checking can be applied have generally resulted in very large interconnected state spaces.

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 318–332, 2001.

A principal result reported in this paper is a set of design rules (Section 3) for development of OO software systems that when translated to representations to which model checking can be applied, yield manageable state spaces. These design rules are the critical initial step in the methodology for integration of formal verification by model checking into OO development processes.

Approach. *The validity and usefulness of design rules and the effectiveness of the integration of formal verification into object-oriented software development can be evaluated only in the context of their application.*

This paper reports a case study in re-engineering the control subsystem for a NASA robotics software system. The goal of the project was to increase the subsystem's reliability. This goal was achieved, as serious logical design errors were discovered, through the application of model-checking. This case study motivates and demonstrates the design rules for "design for verifiability" and the application of formal verification by model checking to a substantial software system.

The robot control subsystem was originally implemented by a conventional development process in a procedural OO programming language (C++) generally following the Booch methodology [1]. In the re-engineering project, a four-step process to obtain a reliable system was planned:

1. Re-implement the control subsystem as an executable specification in the form of an Object-Oriented Analysis (OOA) model (in the Shlaer-Mellor (SM) methodology [26]);

2. Validate this executable specification as thoroughly as possible by testing;

3. Apply model checking to the OOA model to validate its behavior at all possible states of the system;

4. Generate the control software by compilation of the validated and verified OOA model.

The application of the SM OOA method captures the robotic domain entities in terms of classes/objects and relate them using the relationships diagrams. Testing and evaluation of the execution behavior of the constructed system was greatly simplified due to the fact that the OOA classes were represented as a set of attributes, confined to simple types, and that the behavior of the system was state machine specified. Such an OOA model can be viewed as being *designed for testability*. This executable specification can also be translated by a code generation system to C++ code. A commercially supported software system, SES/Objectbench (OB) [25] was used in this step.

OOA models with executable semantics are representations of software system, which should be amenable to model-based verification techniques. An OOA model represents the program at a higher level of abstraction than a conventional programming language. The OOA model partitions the system into well-defined classes. But, attempts to apply model checking to these apparently highly modular OOA models led to intractably large state spaces for the robot control system model. The cause for this problem is suggested by examining hardware systems. In hardware, the "calling" module and "called" module are separated spatially and communicate through a clean interface and a specified protocol. This "spatial modularity" supports divide-and-conquer analytical techniques, as each module can be analyzed in isolation. This is essential for successful model-checking because it resolves a fundamental problem of the *state space*

explosion. The design rules of OOA methods do not enforce the logical equivalent of "spatial modularity" in software. (In software, modularity tends to be "temporal", in the sense that modular subroutines are invoked in succession. This "temporal modularity" does not directly support divide-and-conquer techniques). For example, accessor and mutator methods cause coupling of the states of instances of different classes. The logical equivalent of "spatial modularity" for software is the strong form of name space modularity where the name spaces modules are rigorously disjoint and all interactions among modules are across specified interfaces and follow specified protocols. "Spatial modularity" (strong name space modularity) is consistent with the intent of the OOA approaches of conceptual encapsulation but it is not explicitly considered in most OO design methods. We introduce a set of design rules that constrain the syntactic structure of OOA models to conform to "spatial modularity". The systems become spatially modular (in the hardware sense when system elements can be analyzed in isolation) and support existing verification techniques developed for hardware systems. We applied the *design*

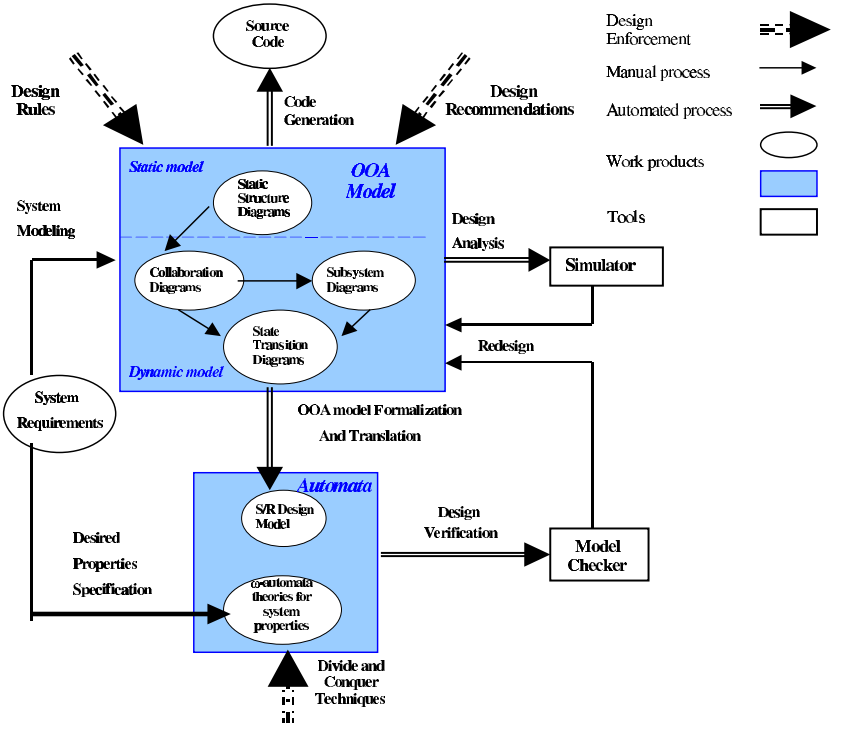


Fig. 1. The OOA-based methodology for the spatial development of software systems

for verifiability rules to a further redesign of the robot controller system. The results are encouraging - we were able to apply the partitioned development, model checking, assume/guarantee reasoning, abstraction techniques [14] developed for hardware systems

to our software system. This powerful combination of techniques helped us to break the computational complexity barrier to the application of verification by model checking to OOA models of software systems. Model checking was accomplished by translation to S/R, an input language of the COSPAN [9] model checker, using the translator reported in [27].

In this paper, we develop a set of design rules for construction of OOA models to which verification by model checking can be practically applied, and we demonstrate the application of the integrated OOA and model-checking methodology for development of software systems.

2 Integration of Model Checking with OO Development

A methodology for integration of OOA and model checking shown in **Figure 1** is presented in [27]. Overall, this model fulfills the need for a sound foundation in rigorous requirements modeling, design analysis, formal verification, and automated code generation. Model checking is applied to SM OOA (xUML) models [26] that have executable semantics specified as state/event machines rather than to programs in conventional programming languages. An automata-based approach to model checking, supported by the COSPAN [9] model checker, is used. The OOA models are automatically translated to automaton models using the OB-SR translator [27]. Predicates over the behavior of the OOA models are mapped to predicates over the automaton models and evaluated by the model checker. This research imposes the structural design rules on the software system reducing its complexity at the design level and thus supports the reuse of the existing model-checking techniques developed for hardware verification.

xUML Notation. Use of OOA models with executable semantics is moving into the mainstream of OO software development. The Object Management Group (OMG) [20] has adopted a standard action language for the Unified Modeling Language (UML) [21]. This action language and SM OOA semantics represented in UML notation define an executable subset of UML (xUML). The OOA representation used in this research is the SM OOA as implemented by the capture and validation environment SES/Objectbench (OB) [25]. We are, on the recommendation of Steve Mellor [private communication] referring to the OOA model we use as xUML.

We utilize a subset of xUML notation suitable for modeling objects, subsystems, their static structure, and their dynamic behavior. *Static structure diagrams* capture conceptual entities as classes with semantics defined by attributes. *Object information diagrams* (OID) describe the classes and relationships that hold between the classes. They graphically represent a design architecture for an application domain and give an abstract description of tasks performed by cooperating objects. *Subsystem relationship diagrams* situate the application domain in relation to its scope, limits, relationships with other domains and main actors involved (scenarios). The *collaboration diagram* is used for graphical representation of the signals sent from one class to another. This representation provides a summary of asynchronous communication between *state/event models* in the system. The state/event model is a set of Moore state machines that consists of a fixed number of concurrently executing finite state machines. The *state transition diagram* graphically represents a state machine. It consists of nodes, representing states

and their associated actions to be performed, and event arcs, which represent transitions between states. The execution of an action occurs after receiving the *signal or event*. A transition table is a list of signals, and "the next" states that are their result. Signals have an arbitrary identifier, a target class, and associated data elements.

Two types of concurrent model execution are supported by xUML: *simultaneous* and *interleaved*. We utilize only the asynchronous interleaved execution model in the OOA models of this research.

COSPAN, an Automaton-based Model Checking Tool. COSPAN [9] allows symbolic analysis of the design model for user-defined behavioral properties. Each such test of task performance constitutes a mathematical proof (or disproof), derived through the symbolic analysis (not through execution or simulation). The semantic model of COSPAN is founded on ω-automata [14]. The system to be verified is specified as an ω-automaton P, the task the system is intended to perform is specified as an ω-automaton T, and verification consists of the automata language containment test $L(P) \subset L(T)$. P is typically given as the synchronous parallel composition of component processes, specified as ω-automata. Asynchronous composition is modeled through nondeterministic delay in the components.

Language containment can be checked in COSPAN using either a symbolic (BDD-based) algorithm or an explicit state-enumeration algorithm.

Systems are specified in the S/R language, which supports nondeterministic, conditional (if-then-else) variable assignments; variables of type bounded integer, enumerated, boolean, and pointer; arrays and records; and integer and bit-vector arithmetic. Modular hierarchy, scoping, parallel and sequential execution, homomorphism declaration and general ω-automata fairness are also available.

3 Design for Verification

An xUML OOA is a natural representation to which to apply model-based verification techniques. The complexity level of the executable OOA models is far less than the procedural language programs to which they are translated. In addition to the finite state representation provided by the OOA techniques, the following features of the OOA methodology reduce the complexity of the system at the design level:

Abstraction of implementation details. Relationships between objects at the OOA level are represented as associations and not as pointers. OOA constructs such as signals in UML express state transitions without reference to the internal states of objects. Separate specification of class models and behavior models separates specification of data from control.

Hierarchical system representation. OOA methods support modular designs and encourage software developers to decompose a system into subsystems, derive interfaces that summarize the behavior of each system, and then perform analysis, validation and verification, using interfaces in place of the details of the subsystems.

"Spatial Modularity" of software systems. The design property which enables verification of hardware systems by model checking is sometimes called "Spatial Modularity". In hardware realized systems functionality is of necessity partitioned into modules which are spatially disjoint. Interaction among these spatially disjoint functional modules

must take place across precisely defined interfaces and follow precisely defined protocols. The spatial partitioning of hardware modules across well-defined static interfaces supports the application of divide-and-conquer techniques, necessary to circumvent the generally infeasible computation problem inherent in model-checking.

The logical equivalent of "spatial modularity" for software is the strong form of name space modularity where the name spaces of all modules are disjoint and all interactions between functional modules are across specified interfaces and follow specified protocols. The strong name space modularity is conceptually consistent with the methodology of separation of concerns advocated by the OOA approaches. It is however is not explicitly specified in any OO design methods.

Structural design rules. We developed a set of design rules and recommendations that constrain the structural design of the OOA models to conform to "spatial modularity". The systems become spatially modular (in the hardware sense where system elements can be analyzed in isolation) and support existing verification techniques developed for hardware systems. These design rules for OOA models are similar to those given for development of truly OO programs in OO procedural languages such as C++.

Design Rule 1. *Write access to attributes of one class by another class must be made through the event mechanism.*

The attributes of a class should be local to the class. Change of values of a class instance should be performed only through the event mechanism. This precludes coupling of internal states of classes.

Design Rule 2. *Attribute values which are shared by multiple classes should be defined in separate class and accessed only through the event mechanism.*

This design rule also avoids coupling of internal states of classes.

Design Rule 3. *Declaration and definition of functional entities must be performed within the same component.*

A component may have dependencies on other components. To prevent the situation when functionality of one component can be changed by other components any logical construct that a component declares should be defined entirely within that component.

Design Rule 4. *Inheritance must be confined to extensions of supertypes. Modification of the behavior of supertypes (overriding of supertype methods) is prohibited.*

This means to follow a meaning of subtyping, along the lines of Liskov's [17]:

"A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of another type (the supertype) plus something extra. What is wanted here is something like the following substitution property: If for every object o1: S there is o2: T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T".

Rule 4 enables reasoning about the correctness of newly derived subtype classes based on the verification results of previously existing subtype classes.

Recommendation. *Creating modular systems the linking between the modules/ subsystems should be minimized.*

Subsystems are fundamentally open systems but for verification must be closed with a definition of the environment in which they will execute. Simulation of the environment behavior is performed by assuming a sequence of events defined on the subsystem's

interface. A minimal number of links between subsystems enables effective definition of the environment used to complete a subsystems definition for verification.

4 The Robot Controller Study

We examine a robotic software used for control of redundant robots [1]. Redundant robots are widely used for sophisticated tasks in uncertain and dynamic environments in life-critical systems. An essential feature for a redundant robot is that an infinite number of robot's joints displacements can lead to a definite wrist (end-effector) position. Failure recovery is one of the examples of redundancy resolution applications: if one actuator fails, the controller locks the faulty joint and the redundant joint continues operating. The general task of the test-case software is to move a robot arm along a specified path given physical constraints (e.g. obstacles, joints angles and end-effector position constraints). The specific task is to choose an optimal arm configuration. This decision-making problem is solved by applying performance criteria [13].

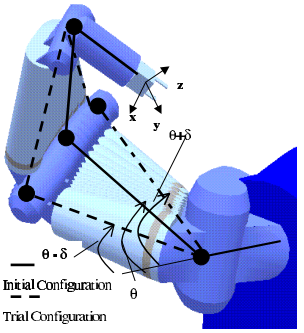


Fig. 2. A part of a redundant robot, demonstrating infinite manipulator configurations for a single end-effector position.

The decision-making method is based on the local explorations and the concept of a joint-level perturbation. Perturbation at the joint level means temporarily changing one or more of the joint angles (joint angle - is the angle between two links forming this joint) either clockwise or counterclockwise. This project focuses on two different exploration strategies: simple and factorial [13]. In the simple exploration we displace one joint at a time and find how it effects the configuration of the robot arm (find all other joint angles) for a given end-effector position. The other perturbation strategy is based on 2^n factorial search. A detail of a redundant robot executing the simple exploration strategy for one of the joints is shown in **Figure 2**, with θ - being a joint angle, and δ - being a displacement.

The original software, OSCAR [13], consisted of a set of robot control algorithms supported by numerous robotic computational libraries, was developed using a conventional approach. To obtain a reliable system we redesigned its control subsystem as an executable specification in the form of a SM OOA (xUML) model.

4.1 Domain Analysis and Modeling

The robotic OOA model includes fifteen basic classes, including their variables and associations.

Classes. In addition to tangible objects (*Arm*, *Joint*, *EndEffector*, *PerformanceCriterion*), incident objects (*TrialConfiguration*, *SearchSpace*, *SimpleSearchSpace*, *FactorialSearchSpace*), specification objects (*Fused Criterion*), and role objects (*Decision Tree*, *OSCAR_Interface*, *Checker*) were derived [26].

[1] Refer to http:/www.robotics.utexas.edu/rrg/glossary for robotic terms

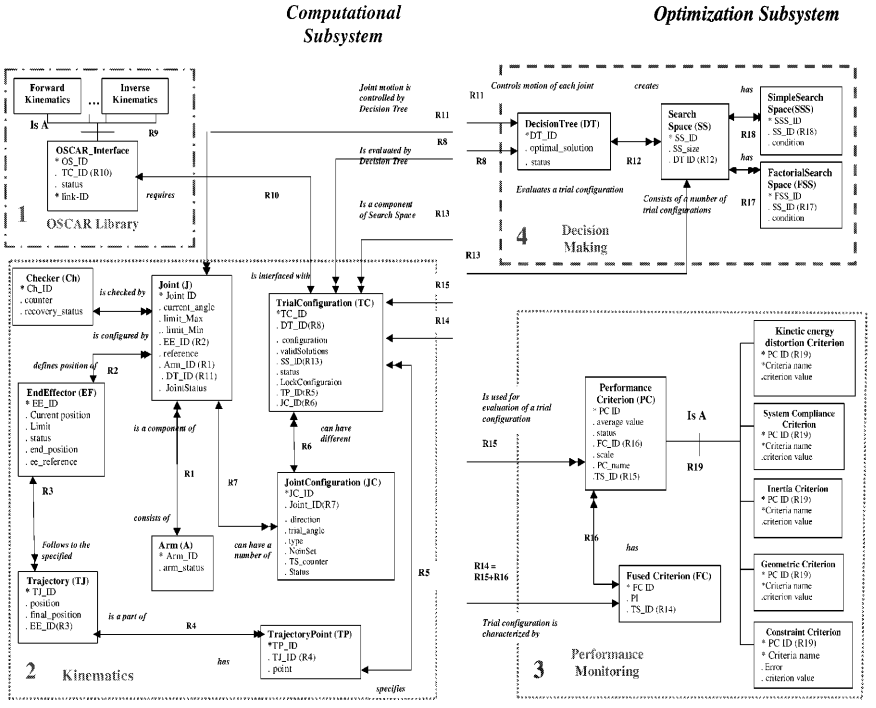


Fig. 3. OID of the Multi-Criteria Decision Support Robotic System

Attributes. The attributes of the *EE* object are illustrated below as an example. *EE_ID* is a key attribute whose value uniquely distinguishes each instance of an *EE* object. *Current_position* and *Limit* are descriptive attributes that provide facts intrinsic to the *EE* object. For example, *Current_position* is a vector specifying positions (*x, y, z*) and orientation angles (α, β, γ) of the *EE*. *Status*, *end_position* and *ee_reference* are so called naming attributes which provide facts about the arbitrary labels carried by each instance of an object. The domain of the naming attributes is specified by enumeration of all possible values that the attribute can take on. For example, *end_position* domain is (0,1) which values reflect if the *EE* reached the final destination while moving along the specified trajectory path.

Associations. The executable model is defined using two types of the relationships: binary (those in which objects of two different types participate) and higher-order supertype-subtype (those when several objects have certain attributes in common which are placed in the supertype object). For example, one-to-many binary *Arm-Joint* relationship states that a single instance of an Arm object consists of many instances of a *Joint* object. An example of supertype-subtype relationships is a *PerformanceCriterion-*

ConstraintCriterion relationship. In this construct one real-world instance is presented by the combination of an instance of the supertype and an instance of exactly one subtype.

Robotic Decision-Support Domain Architecture. The application domain architecture was divided into *computational* and *optimization* subsystems. The input for the optimization subsystem is one or more trial arm configurations from which the optimization system will either select the best one or provide the computational system with suggestions on what an optimal arm configuration should be.

The *computational* subsystem includes kinematics algorithms and interfaces to the computational libraries of the OSCAR system. There are two methods for the computational system to define a base point of the optimization search. The first method is to calculate an *EndEffector* (*EE*) position given initial *Joint* (*J*) angles of all joints and, thus, find an initial arm configuration. The second method depicts an *EE* position as a new base point from the *Trajectory* path specified by the user.

A collaboration diagram of the abstracted *Kinematics* unit which verification results we discuss in the next section is represented in Figure 6. The control algorithm starts with defining an initial end-effector position given the initial joint angles. This is done by solving a forward kinematics problem [13]. The next step is to get a new end-effector position from a predefined path. The system calculates the joint angles for this position, providing the solution of the inverse kinematics problem [13] and configures the arm. At each of the steps described above, a number of physical constraints has to be satisfied. The constraints include limits on the angles of joints. If a joint angle limit is not satisfied, a *fault recovery* is performed. The faulty joint is locked within the limit value. Then, the value of the angle of another joint is recalculated for the same end-effector position. If the end-effector position exceeds the limit, the algorithm registers the undesired position, which serves as a flag to stop the execution. A *Checker* class controls the joints that pass or fail the constraints check. If all the joints meet the constraints, the *Checker* issues the command to move the end-effector to a new position. Otherwise it either starts a fault recovery algorithm or stops execution of the program (if fault recovery is not possible).

The *optimization* subsystem implements the decision-making strategy by applying decision-making techniques identifying a solution to the multi-criteria problem. It builds a *SearchSpace* (*SS*), which generates sets of *JointConfigurations* (*JC*) around a base point supplied by the computational subsystem. *JC* instances initiate creation of *TrialConfiguration* (*TC*) instances that normalize a robot arm configuration for any perturbed joint. A *DecisionTree* (*DT*) selects the best *TC* given a set of *PerformanceCriteria* (*PC*) and a number of physical constraints that are globally defined by the user. The found solution serves as the next base point for another pattern of local exploration. The search stops when no new solutions are found. The system returns control to the computational subsystem which changes the position of the *EE* following the specified trajectory and determines a new base point for the search.

4.2 Compliance to the Design Rules

During the construction of the robot control design we followed the design rules specified in Section 3.

Rule 1: All updates of the attribute values were done through t the event mechanism.

Rule 2: After the design system was completed and validated by simulation, a separate object *Global* that contained all the global variables as its attributes was created and used during verification and code generation.

Rule 3: We restricted the design to insure that all functional components are fully self-contained.

Rule 4: The users of the developed robotic framework are allowed to add new elements to the developed architecture. Specifically, new performance criteria can be added to the architecture. These additions are subtype classes and, in order to satisfy the fourth rule, it is required that they have a semantic relationship with their supertype classes. Other words, inheritance is restricted to a purely syntactic role: code reuse and sharing, and module importation.

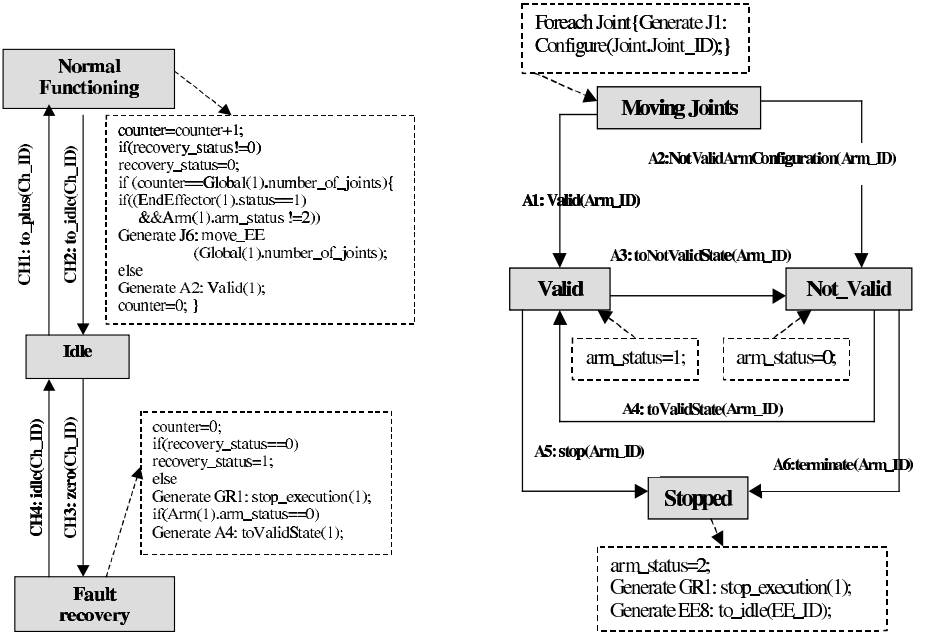


Fig. 4. State Transition Diagram of the *Checker* (left) and *Arm* (right) objects

Recommendation (System decomposition): As it can be seen in **Figure 3** the architecture can be represented as a collection of basic robotics functional units. These functional components (*OSCAR_Library*, *Kinematics*, *Performance Monitoring* and *Decision Making*) are depicted by dashed lines. Each functional unit contains a substantial proportion of components that do not depend on other units.

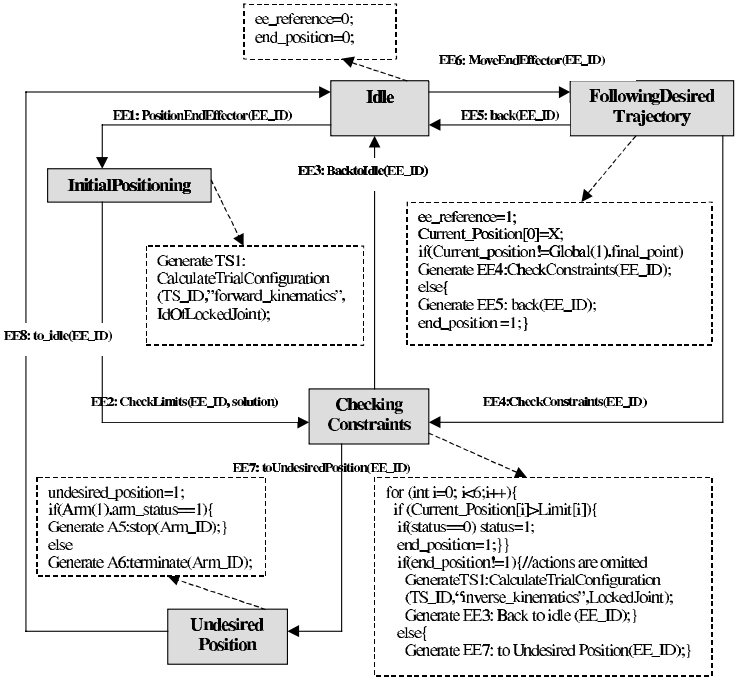


Fig. 5. State Transition Diagram of the *EndEffector* object

4.3 OOA Model Validation and Formal Verification

The OOA model was validated by simulation. Several serious error or defects in the original design and in the original versions of the OOA model were identified and corrected. Space precludes us from describing the validation process and its results. Details can be found in [22].

We checked a collection of safety and guarantee requirements specifying the coordinated behavior of the robot control processes. We focused on the control intensive algorithms of the *Kinematics* unit and abstracted the calculations that were irrelevant to the actual robot control. Specifically, the *Trajectory*, *TrajectoryPoint* and *JointConfiguration* classes used for storage of the predefined trajectory paths and the possible arm configurations as well as calculations that were done through the interface with the OSCAR Libraries in the original OOA design were substituted with nondeterministic assignments of natural numbers. In fact, in this paper we present an instance of the robot functionality when the robot arm is moving only in the horizontal, i.e. x direction, which value is assigned nondeterministically in the checked model.

We define and discuss here the properties that did not hold during the verification. The properties and their descriptions are given in **Table 1**. The properties are encoded in a query language of COSPAN. The query variables are declared in terms of state predicates appearing in the state transition diagrams of the objects of the *Kinematics* unit. The following declarations are used in Table 1: p - declares the *abort_var* variable of

the *Global* class; *q* - declares the *undesired_position* variable of the *EE* class; *r* - declares the *ee_reference* variable of the *EE* class; *s* - declares the *recovery_status* variable of the the *Checker* class; *t* - declares the *end_position* variable of the *EE* class; *v* - declares the *number_of_joints* variable of the *Global* class.

Figure 4, 5 schematically represent the lifecycles of the robot control processes (some actions are omitted due to the space limitations of the paper). For example, the state *UndesiredPosition* and the variables *undesired_position*, *ee_reference* of the *EE* class appear in Figure 5.

Verification found a number of errors in the robot control algorithms. The failure of the Property 1 indicated that in some cases the system does not terminate its execution as specified. The failure of the property 3 that was aimed to check if system terminates properly confirmed this fact. We learned that an error in the fault resolution algorithm caused this problem. We will remind the reader that the fault recovery procedure is activated if one of the robot joints does not satisfy the specified limits. In fact, if during the process of fault recovery some of the newly recalculated joint angles do not satisfy the constraints in their turn, then another fault recovery procedure is called. Analysis of the counterexample provided by COSPAN for Property 3 indicated that a mutual attempt was made for several faulty joints to recompute the joint angles of other joints while not resolving the fault situation.

Table 1. Verification properties

N	Property	Robotic Description	Formal Description
1	EventuallyAlways(p=1)	Eventually the robot control terminates	Eventually permanently p=1
2	AfterAlwaysUntil(q=1, r=1, p=1)	If the *EE* reaches an undesired position than the program terminates prior to a new move of the *EE*	At any point in the execution if q=1 than it is followed by r=1 until p is set to 1
3	AlwaysUntil(p=0, t=1 OR (s=1 AND v=1))	The program terminates when it either completes the task or reaches the state where there is no solution for the fault recovery	p=0 holds at any execution of the program until occurrence of either t=1 or the combination of s=1 and v=1

Another error that was found during verification of Property 2 indicated a problem of coordination between the *Arm* and *Checker* processes. The original design assumed sequential execution. In fact, it was expected that the *arm_status* variable of the *Arm* process would be repeatedly updated before the *Checker* process would initiate a command to move the *EE* to a new position. A concurrent interaction between the processes led to the situation where the *Checker* process could issue the command based on an out-of-date value of the *arm_status* variable. This was the reason for Property 2 to fail.

The errors found by model checking were not discovered either during the conventional testing performed by the developers of the original code or during the validation by simulation of the formalized design. In order to correct these errors a redesign of both the original system and the OOA model was required. Figure 6 provides the original and the modified state transition diagrams of the *Kinematics* unit demonstrating the

design changes which we made in order to correct the found errors. We introduced a new class called Recovery, whose functionality provides a correct resolution of the fault recovery situation described above. Additionally we added exchange messages between the processes *Arm* and *Checker* that fixed the coordination problem reported earlier.

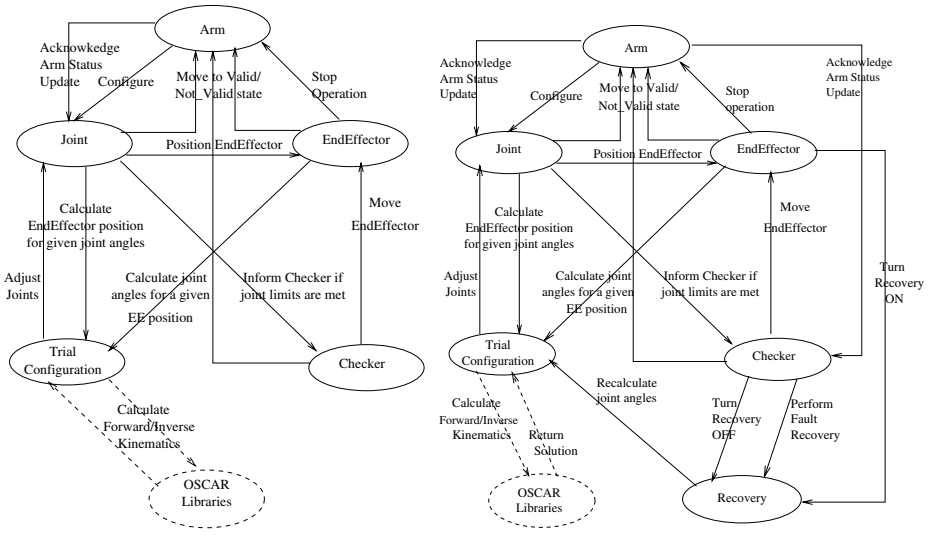


Fig. 6. Collaboration diagrams of the original (left) and modified (right) *Kinematics* unit

It is interesting to note that concurrently with this project, we examined the possibility of integrating testing into the model checking process. The SM OOA executable specification of the robot control system was used as a test-bed for that project. Specifically, an abstracted version of the *Kinematics* unit was manually translated into Promela, the input language of the SPIN [12] model checker. PET [8], an interactive testing tool that supports visual representation of data, was used to establish the conformance between the source code and the code accepted by the model checker. SPIN verification results are presented in [23]. Design errors found using SPIN were corrected and in this paper we use the corrected robot control system. The failure of the fault tolerance algorithm, however, is demonstrated in both projects. The fact that we received identical verification results using different model checking tools for property 3 confirms the validity of the verification results.

4.4 Robotic System Engineering

Given the target system specifications, the validated and verified architecture, and the target system configuration parameters, an instance of the target robotic system was automatically generated using SES/CodeGenesis system [24]. C++ source code that supports the implementation of the developed architecture can be found at www.robotics.utexas.edu/rrg/organization/dual_arm/research/ROOA/.

5 Conclusions and Related Work

This paper gives a feasibility demonstration for the application of verification by model checking to a substantial control intensive application developed in a commercially supported and widely used OO development process. The results of the demonstration are highly encouraging. Verification of significant behavioral properties of the robot control subsystem were carried out. The importance of verification to OOA model design and development has been shown. Design rules leading to xUML OOA models to which verification by model checking can be practically applied have been proposed and applied.

Previous work on application of model checking to software systems has mainly been either to software systems written in procedural languages or to abstract models extracted from programs in procedural languages. Feaver [11] targets software systems written in C while [2], [3] focus on applying model checking on SDL programs. Havelund and Pressburger [10] apply model checking to Java programs. Corbett, et.al [5] extract finite state machines from Java programs to which to apply model checking. The results of verification of a safety critical railroad control system which complexity is comparable to our test-case study are presented in [7].

Model-checking has been also applied to verification of concurrently executing state/event machines. Lind-Nielsen, et al. [16] applied SMV [18] for verification of hardware systems represented by VisualState state machines. Dependency analysis was used to decompose a large but naturally spatially modular systems. Chan, et al. [6] verified a complex aircraft collision software. They reported that their ad hoc solutions for the manual system partitioning frequently caused invalid results. None approaches the issues of the system redesign prior to model-checking.

Design guidelines for constructing testable and maintainable programs in object-oriented procedural languages have been proposed and discussed by a number of researchers [15], [17]. Moors [19] has proposed similar design criteria for communication protocols. However, there is no an effort known to us that would address a problem of developing the OOA design rules that support resolution of the state-explosion problem at the design level.

Acknowledgement. This research was partially supported by the Robotics Research Group of the University of Texas at Austin.

References

1. Booch, G., *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, Redwood City, CA (1994)
2. Bounimova, E., Levin, V., Basbugoglu, O., and Inan, K., A Verification Engine for SDL Spec. Of Comm. Protocols, *In Proc. of the 1st Symp. on Computer Networks*, Istanbul, Turkey, (1996) 16-25
3. Bosnacki, D., Damm, D., Holenderski, L.,and Sidorova, N., Model checking SDL with Spin, *In Proc. of TACAS2000, Berlin, Germany*, (2000) 363-377
4. Clarke, E.M., and Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic, *Workshop on Logic of Programs, Yorktown Heights, NY.* LNCS, Vol. 131, (1981) 52-71

5. Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Pasareanu, C., Bandera: Extracting finite-state models for Java source code, *In Proc. of 22nd ICSE* (2000)
6. Chan, W., Anderson, R., Beame, P., Burns, S., Modugno, F., Notkin, D., Reese, J., Model Checking Large Software Specifications, *In Proc. of IEEE Transaction on Software Engineering* (1998) 498-519
7. Gnesi, S., Lenzini, G., Abbaneo, C., Latella, D., Amendola, A., Marmo, P., An Automatic SPIN Validation of a Safety Critical Railway Control System, *In Proc. of Int. Conf. on Dependable Systems and Networks*, (2000) 119-124
8. Gunter, E., and Peled D., Path Exploration Tool, *In Proc. of TACAS 1999, Amsterdam, The Netherlands* (1999) 405-419
9. Hardin R., Har'El, Z., and Kurshan, R.P., COSPAN, *In Proc., CAV'96*, LNCS, Vol. 1102, (1996) 423-427
10. Havelund, K., and Pressburger, T., Model Checking Java Programs Using Java PathFinder, *In Proc. 4'th SPIN workshop* (1998)
11. Holzmann, G., and Smith, M., Feaver: Automating software feature verification, *Bell Labs Technical Journal*, Vol. 5, 2, (2000) 72-87
12. Holzmann, G., The Model Checker SPIN, *IEEE Trans. on Software Engineering*, Vol. 5(23), (1997) 279-295
13. Kapoor, C., and Tesar, D.: A Reusable Operational Software Architecture for Advanced Robotics (OSCAR), The University of Texas at Austin, Report to DOE, Grant No. DE-FG01 94EW37966 and NASA Grant No. NAG 9-809 (1998)
14. Kurshan, R., *Computer-Aided Verification of Coordinating Processes - The Automata-Theoretic Approach*, Princeton University Press, Princeton, NJ (1994)
15. Lano, K., *Formal Object-Oriented Development*, Springer (1997)
16. Lind-Nielsen, J, Andersen H., R., etc., Verification of large State/Event Systems using Compositionality and Depenedency Analysis, *In Proc. of TACAS'98, Portugal* (1998) 201-216
17. Liskov, B., Data Abstraction and Hierarchy, *In Proc. of OOPSLA conference* (1987)
18. McMillan, K. *Symbolic Model Checking*, Kluwer (1993)
19. Moors, T., Protocol Organs: Modularity should reflect function, not timing, *In Proc. OPENARCH98*, (1998) 91-100
20. Object Management Group (OMG), *Action Semantic for the UML*, OMG (2000)
21. Rumbaugh, J., Jacobson, I. and Booch, G., *The Unified Modeling Language Reference Manual*, Object Technology Series, Addison-Wesley (1999)
22. Sharygina, N., and Browne, J., Automated Rob. Decision Support Software Reverse Engineering, Tech. Rep., The Univ. of Texas at Austin, Robotics Research Croup (1999)
23. Sharygina, N., and Peled, D., A Combined Testing and Verification Approach for Software Reliability, *In Proc. of FME2001* (to appear), Berlin (2001)
24. SES Inc., *CodeGenesis User Reference Manual*, SES Inc. (1998)
25. SES inc., *ObjectBench Technical Reference*, SES Inc. (1998)
26. Shlaer, S., and Mellor, S., *Object Lifecycles: Modeling the World in States*, Prentice-Hall, NJ (1992)
27. Xie, F., Levin, V., Browne, J., Integrating model checking into object-oriented software development process, Techn.Rep., University of Texas at Austin, Comp. Science Dept. (2000)

Specification and Analysis of the AER/NCA Active Network Protocol Suite in Real-Time Maude

Peter Csaba Ölveczky[1,4], Mark Keaton[2], José Meseguer[1], Carolyn Talcott[3], and Steve Zabele[2]

[1] Computer Science Laboratory, SRI International, Menlo Park, CA 94025
[2] Litton-TASC Inc., Reading, MA 01867
[3] Computer Science Department, Stanford University, Stanford, CA 94305
[4] Department of Informatics, University of Oslo, Norway

Abstract. This paper describes the application of the Real-Time Maude tool and the Maude formal methodology to the specification and analysis of the AER/NCA suite of active network multicast protocol components. Because of the time-sensitive and resource-sensitive behavior and the composability of its components, AER/NCA poses challenging new problems for its formal specification and analysis. Real-Time Maude is a natural extension of the Maude rewriting logic language and tool for the specification and analysis of real-time object-based distributed systems. It supports a wide spectrum of formal methods, including: executable specification; symbolic simulation; and infinite-state model checking of temporal logic formulas. These methods complement those offered by finite-state model checkers and general-purpose theorem provers. Real-Time Maude has proved to be well-suited to meet the AER/NCA modeling challenges, and its methods have been effective in uncovering subtle and important errors in the informal use case specification.

1 Introduction

This paper describes the application of the Real-Time Maude tool [14] and the Maude formal methodology [4] to the specification and analysis of the AER/NCA suite of active network communication protocol components [8,1] which collectively implement a scalable and reliable multicast capability using active elements in the network. Being a very advanced and sophisticated suite of protocols that run in a highly distributed and modular fashion, the AER/NCA suite poses challenging new problems for formal specification and analysis including:

- Time-sensitive behavior, including delay, delay estimation, timers, ordering, and resource contention;
- Resource-sensitive behavior, including capacity, latency, congestion/cross-traffic, and buffering;

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 333–347, 2001.

- Both performance and correctness are critical metrics;
- Composability issues: modeling and analyzing both individual protocol components and their aggregate behavior; and supporting reuse for developing alternative protocols.

Maude is a language and high-performance system based on rewriting logic [3]. As such it naturally supports specification and analysis of object-based distributed systems by supporting a wide spectrum of formal methods [4,11], including executable specification; symbolic simulation; model checking; and formal proof. Real-Time Maude naturally extends Maude to support the above formal methodology to distributed real-time and hybrid systems [14,12].

Real-Time Maude has proved to be well-suited to meet the above challenges. The active network and performance aspects have been naturally addressed by the flexibility of the Maude's distributed object model that made it easy to include active elements and resources as objects. The time- and resource-sensitive behavior is expressed naturally by timed rewrite rules. The composability issues were well addressed by Maude's support for multiple class inheritance.

The starting point of the formal specification effort was an informal specification consisting of a set of use cases. Although use cases are widely used as a software design technique, the experience gained from the present work indicates that they are not well suited for modeling complex distributed systems. To understand the system behavior, state transition diagrams had to be developed by the protocol designers. The Maude specification provided a natural formalization of the informal state transition diagrams and followed closely the designers' intuitions. In hindsight, it seems clear that, for distributed applications of this kind, the executable state-transition style of the Maude specification is a much more effective starting point for an implementation than use cases.

The Maude formal methodology complements other formal methods approaches such as model checking tools [7,2,9,18] and general purpose theorem provers [16, 17,6]. It provides a flexible middle ground extending the advantages of model checking to a wide range of infinite state systems. Furthermore, the simple formal semantics of rewriting logic and the underlying equational and rewriting techniques provide a natural basis for a range of automated and interactive deduction techniques.

2 Specifying and Analyzing Real-Time Systems in Rewriting Logic

In rewriting logic [10] distributed systems are specified by *rewrite theories* of the form (Σ, E, R), with (Σ, E) an equational theory specifying the system's state space as an algebraic data type, and R a collection of rewrite rules specifying the system's local transitions. This specification style can be specialized to distributed real-time and hybrid systems by using *real-time rewrite theories* [15]. The Real-Time Maude language and tool [14,12] can then be used for specifying, simulating, and analyzing such systems.

2.1 Real-Time Rewrite Theories

In [15] we have proposed modeling real-time and hybrid systems in rewriting logic as *real-time rewrite theories*. These are rewrite theories containing:

- a specification of a *Time* data type specifying the time domain;
- a sort *System* with no subsorts, and a free constructor $\{\_\} : State \to System$ (for *State* the sort of the global state) with the intended meaning that $\{t\}$ denotes the whole system, which is in state t;
- *instantaneous rewrite rules* that model instantaneous change and are assumed to take zero time; and
- *tick (rewrite) rules* that model the elapse of time on a system, and have the form

$$[l] : \{t(x_1, \ldots, x_n)\} \stackrel{\tau_l(x_1,\ldots,x_n)}{\longrightarrow} \{t'(x_1, \ldots, x_n)\} \textbf{ if } cond,$$

  with $\tau_l(x_1, \ldots, x_n)$ a term of sort *Time* denoting the rule's *duration*. The use of the operator $\{\_\}$ in the tick rules ensures uniform time advance by the global state always having the form $\{t\}$.

In [15] we have shown that a wide range of models of real-time and hybrid systems can be expressed quite naturally and directly as real-time rewrite theories. We have also shown in [15] that real-time rewrite theories can be reduced to ordinary rewrite theories by adding an explicit clock to the global state in a way that preserves all their expected properties. This transformation introduces a new constructor $\langle \_, \_ \rangle : System\ Time \to ClockedSystem$, replaces each tick rule of the form $[l] : \{t\} \stackrel{\tau_l}{\longrightarrow} \{t'\}$ **if** *cond* with a rule of the form $[l] : \langle \{t\}, x \rangle \longrightarrow \langle \{t'\}, x + \tau_l \rangle$ **if** *cond* (for x a new variable), and leaves the rest of the theory unchanged.

2.2 Real-Time Maude

The Real-Time Maude specification language and analysis tool [12,14] is built on top of the rewriting logic language Maude [3]. Real-Time Maude supports the specification of real-time rewrite theories in *timed modules* and *object-oriented timed modules*, which are transformed into equivalent Maude modules.

2.3 Specifying Concurrent Objects

Real-Time Maude extends Full Maude [5,3]. We recall how concurrent objects are specified in *object-oriented modules* in Full Maude. A class declaration

`class` C `|` att_1 `:` s_1 `, ... ,` att_n `:` s_n `.`

declares a class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C is represented as a term `<` O `:` C `|` att_1 `:` val_1`,` ... `,` att_n `:` val_n `>`, where O is the object's name or identifier, and where val_1 to val_n are the current values of the attributes att_1 to att_n. In a concurrent object-oriented system, the state, which is usually called a *configuration*, has typically the structure of a multiset made up of objects and messages, and where multiset union is denoted by an associative and commutative juxtaposition operator (empty syntax). The dynamic behavior of concurrent object systems is axiomatized by specifying each of its concurrent transition patterns by a rewrite rule. For example, the rule

```
rl [l] : m(O,w) < O : C | a1 : x, a2 : y, a3 : z > =>
                < O : C | a1 : x + w, a2 : y, a3 : z > m'(y,x)
```

defines a (family of) transition(s) in which a message `m` having arguments `O` and `w` is consumed by an object `O` of class `C`, with the effect of altering the attribute `a1` of the object and of generating a new message `m'(y,x)`. By convention, attributes, such as `a3` in our example, whose values do not change and do not affect the next state of other attributes need not be mentioned in a rule. Attributes like `a2` whose values influence the next state of other attributes or the values in messages, but are themselves unchanged, may be omitted from righthand sides.

2.4 Specification of Object-Oriented Real-Time Systems

In Real-Time Maude, tick rules of the form $[l]: \{t\} \xrightarrow{\tau_l} \{t'\}$ **if** *cond* are written with the syntax

```
crl [l] : {t} => {t'} in time τ_l if cond .
```

(and with similar syntax for unconditional rules). We recall here some of the techniques outlined in [15] for specifying object-oriented real-time systems used in this case study. Such systems are specified as *timed object-oriented modules* in Real-Time Maude. The single tick rule in the AER/NCA specification is

```
var OC : ObjConf .
crl [tick] : {OC} => {delta(OC, mte(OC))} in time mte(OC)
                     if mte(OC) =/= INF and mte(OC) =/= 0 .
```

The use of the variable `OC` of sort `ObjConf` (denoting configurations consisting of objects only) requires that the global state only consists of objects when the `tick` rule is applied, and therefore forces messages to be treated without delay, because the above rule will not match and therefore time will not advance when there are messages present in the state.

The function `delta` models the effect of time elapse on a state, the function `mte` denotes the *maximal time elapse* possible before some instantaneous action must be taken, and `INF` is an infinity value. The functions `delta` and `mte` distribute over the objects in a configuration as follows:

```
op delta : Configuration Time -> Configuration .
op mte : Configuration -> TimeInf .

vars NECF NECF' : NEConfiguration .  var R : Time .
eq delta(none, R) = none .
eq delta(NECF NECF', R) = delta(NECF, R) delta(NECF', R) .
eq mte(none) = INF .
eq mte(NECF NECF') = min(mte(NECF), mte(NECF')) .
```

To completely specify these functions, they must then be defined for single objects as illustrated in Section 4.3.

2.5 Rapid Prototyping and Formal Analysis in Real-Time Maude

The Real-Time Maude analysis tool supports a wide range of techniques for formally analyzing timed modules which we summarize below.

Rapid Prototyping. The Real-Time Maude tool transforms timed modules into ordinary Maude modules that can be immediately executed using Maude's default interpreter, which simulates one behavior—up to a given number of rewrite steps to perform—from a given initial state. The tool also has a default *timed* execution strategy which controls the execution by taking the elapsed time in the rewrite path into account.

Model Checking. Real-Time Maude provides a variety of search and model checking commands for further analyzing timed modules by exploring all possible behaviors—up to a given number of rewrite steps, duration, or satisfaction of other conditions—that can be nondeterministically reached from the initial state. In particular, the tool provides model checking facilities for model checking certain classes of real-time temporal formulas [14]. In this paper we will model check temporal properties of the form p $\mathbf{UStable}_{\leq r}$ p', where p and p' are *patterns*, and $\mathbf{UStable}_{\leq r}$ is a temporal "until/stable" operator. A pattern is either the constant `noTerm` (which is not matched by any term), the constant `anyPattern` (which is matched by any term), a term (possibly) containing variables, or has the form $t(\overline{x})$ `where` $cond(\overline{x})$. The temporal property p $\mathbf{UStable}_{\leq r}$ p' is satisfied by a real-time rewrite theory with respect to an initial term t_0 if and only if for each infinite sequence and each non-extensible finite sequence

$$\langle t_0, 0\rangle \longrightarrow \langle t_1, r_1\rangle \longrightarrow \langle t_2, r_2\rangle \longrightarrow \cdots$$

of one-step sequential ground rewrites [10] in the transformed "clocked" rewrite theory (see Section 2.1), there is a k with $r_k \leq r$ such that t_k matches p', and t_i matches p for all $0 \leq i < k$, and, furthermore, if t_j matches p' then so does t_l for each $l > j$ with $r_l \leq r$. That is, each state in a computation matches p until p' is matched for the first time (by a state with total time elapse less than or equal to r), and, in addition, p' is matched by all subsequent states with total time elapse less than or equal to r.

Application-Specific Analysis Strategies. A Real-Time Maude specification can be further analyzed by using Maude's reflective features to define application-specific analysis strategies. For that purpose, Real-Time Maude provides a library of strategies—including the strategies needed to execute Real-Time Maude's search and model checking commands—specifically designed for analyzing real-time specifications. These strategies are available in the Real-Time Maude module `TIMED-META-LEVEL`, allowing the strategy library to be reused in modules importing `TIMED-META-LEVEL`. Section 5.2 gives an example of an application-specific strategy which was easily defined (in 35 lines of Maude code) by reusing key functions from `TIMED-META-LEVEL`.

3 The AER/NCA Protocol Suite

The AER/NCA protocol suite [1,8] is a new and sophisticated protocol suite for reliable multicast in active networks. The suite consists of a collection of composable protocol components supporting active error recovery (AER) and nominee-based congestion avoidance (NCA) features, and makes use of the possibility of having some processing capabilities at "active nodes" between the sender and the receivers to achieve scalability and efficiency.

The goal of reliable multicast is to send a sequence of data packets from a sender to a group of receivers. Packets may be lost due to congestion in the network, and it must be ensured that each receiver eventually receives each data packet. Existing multicast protocols are either not scalable or do not guarantee delivery. To achieve both reliability and scalability, Kasera et al. [8] have suggested the use of *active services* at strategic locations inside the network. These active services can execute application-level programs inside routers, or on servers co-located with routers along the physical multicast distribution tree. By caching packets, these active services can subcast lost packets directly to "their" receivers, thereby localizing error recovery and making error recovery more efficient. Such an active service is called a *repair server*. If a repair server does not have the missing packet in its cache, it aggregates all the negative acknowledgments (NAKs) it receives, and sends only one request for the lost packet towards the sender, solving the problem of feedback implosion at the sender.

3.1 Informal Description of the Protocol

The protocol suite consists of the following four composable components:

- The *repair service (RS)* component deals with packet losses and tries to ensure that each packet is eventually received by each receiver in the multicast group.
- *Rate control (RC):* The loss of a substantial number of packets indicates over-congestion due to a too high frequency in the sending of packets. The rate control component dynamically adjusts the rate by which the sender

sends new packets, so that the frequency decreases when many packets are lost, and increases when few packet losses are detected.
- *Finding the nominee receiver (NOM):* The sender needs feedback about discovered packet losses to adjust its sending rate. However, letting *all* receivers report their loss rates would result in too many messages being sent around. The protocol tries to find the "worst" receiver, based on the loss rates and the distance to the sender. Then the sender takes only the losses reported from this *nominee* receiver into account when determining the sending rate.
- *Finding round trip time values (RTT):* To determine the sending rate, the nominee, and how frequently to check for missing packets, knowledge about the various *round trip times* (the time it takes for a packet to travel from a given node to another given node, and back) in the network is needed.

These four components are defined separately, each by a set of use cases, in the informal specification [1,12], and are explained in [12,8]. In our formal specification the rewrite rules closely correspond to the use cases.

4 Formal Specification of the AER/NCA Protocol Suite in Real-Time Maude

We summarize in this section the Real-Time Maude specification of the AER/NCA protocol suite, which is described in its entirety in [12,13]. Although the four protocol components are closely inter-related, it is nevertheless important to analyze each component separately, as well as in combination.

4.1 Modeling Communication and the Communication Topology

We abstract away from the passive nodes in the network, and model the multicast communication topology by the multicast distribution tree which has the sender as its root, the receivers in the multicast group as its leaf nodes, and the repair servers as its internal nodes. The appropriate classes for these objects are defined as follows, where the sorts `OidSet` and `DefOid` denote, respectively, sets of object identifiers and the object identifiers extended with a default value `noOid`:

```
class Sendable | children : OidSet .
class Receivable | repairserver : DefOid .
class Sender .   subclass Sender < Sendable .
class Receiver .   subclass Receiver < Receivable .
class Repairserver .   subclass Repairserver < Sendable Receivable .
```

Packets are sent through links, which model edges in a multicast distribution tree. The time it takes for a packet to arrive at a link's target node depends on the size of the packet, the number of packets already in the link, and the speed and propagation delay of the link. All these factors affect the degree of congestion and must be modeled to faithfully analyze the AER/NCA protocol. The class

`LINK` models all these aspects. The attempt to enter a packet p into the link from a to b is modeled by the message `send`(p, a, b). This message is treated by the link from a to b by discarding the packet if the link is full, and otherwise by delivering it—after a delay corresponding to the transmission delay—by sending the message p `from` a `to` b to the global configuration, where it should then be treated by object b.

4.2 The Class Hierarchy

The Real-Time Maude specification is designed, using multiple class inheritance, so that each of the four protocol components RTT, NOM, RC, and RS can be executed separately as well as together in combination. Figure 1 shows the class hierarchy for sender objects, which allows for maximal reuse of transitions which have the same behavior when a component is executed separately and when it is executed together with the other components. The class hierarchies for repair servers and receivers are entirely similar.

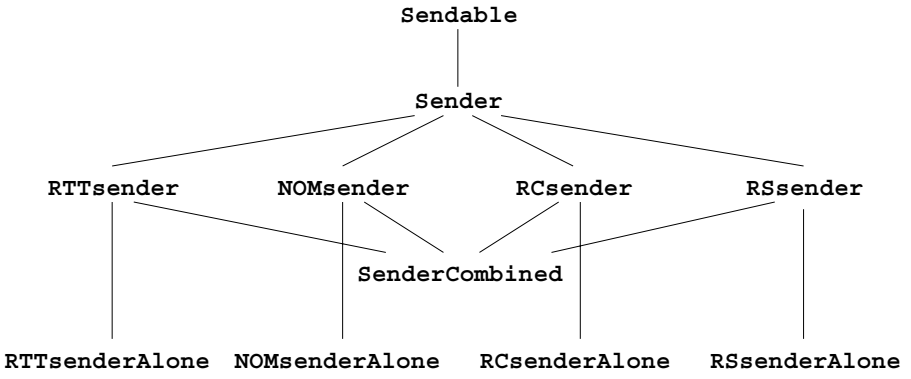


Fig. 1. The sender class hierarchy.

4.3 Specifying the Receiver in the Repair Service Protocol

To exemplify the Real-Time Maude specification style, we present some parts of the specification of the receiver objects in the RS protocol. The receiver receives data packets and forwards them to the receiver *application* in increasing order of their sequence numbers. Received data packets that cannot be forwarded to the application because some data packets with lower sequence numbers are missing, are stored in the `dataBuffer` attribute, and the smallest sequence number among the non-received data packets is stored in the `readNextSeq` attribute. When the receiver detects the loss of a data packet, it waits a small amount of time (in

case some of its "siblings" or its repair server also have detected the loss) before sending a NAK-request for the lost packet to its repair server. The repair server then either subcasts the data packet from its cache or forwards the request upstream. The receiver retransmits its request for the missing data packet if it does not receive a response to the repair request within a reasonable amount of time.

We store, for each missing data packet, the information about the recovery attempts for the missing data packets in a term

$$\texttt{info}(seqNo, supprTimer, retransTimer, NAKcount),$$

where *seqNo* is the sequence number of the data packet, *supprTimer* is the value of the suppression timer for the data packet (this value is either the value `noTimeValue` when the timer is turned off, or the time remaining until the timer expires), *retransTimer* is the value of the retransmission timer of the data packet, and *NAKcount* is the NAK count of the data packet, denoting how many times a repair for the data packet has been attempted. Elements of a sort `DataInfo` are multisets of `info` terms, where multiset union is denoted by an associative and commutative juxtaposition operator.

The receiver class in the RS component is declared as follows:

```
class RSreceiver |
      fastRepairFlag : Bool,
      readNextSeq : NzNat,         *** first missing data packet
      retransTO : Time,            *** time before resending NAK packet
      dataBuffer : MsgConf,        *** buffered dataPackets
      ...
      dataInfo : DataInfo .        *** store info about repairs
subclass RSreceiver < Receiver .

class RSreceiverAlone .   subclass RSreceiverAlone < RSreceiver .
```

As an example of the modeling of the use cases in the informal specification, we show the use case and corresponding rule that describes what happens when the suppression timer for a missing data packet expires. That is, when the second parameter of an `info`-term is `0`. The use case in the informal AER/NCA specification is given as follows:

```
B.5   This use case begins when the NAK suppression timer for a missing
      data packet expires. The following processing is performed (seq
      is the sequence number of the missing data packet):

    if ((data packet seq is currently buffered) OR (seq < readNextSeq))
    { End Use Case }
    if (NAK count for data packet seq > 48)
    { Error, connection is broken, cannot continue }
    Unicast a NAK packet for data packet seq with the receiver's NAK
      count and fastRepairFlag to repairServer
    Start a NAK retransmission timer for data packet seq with a
      duration of retransTO
```

This use case is modeled in Real-Time Maude as follows:

```
vars Q Q' : Oid .  vars NZN NZN' : NzNat .  var X : Bool .
var MC : MsgConf .  vars DI DI' : DataInfo .  var N : Nat .
var DT : DefTime .  var CF : Configuration .  var R : Time .
op ERROR : -> Configuration .

rl [B5] :
   {< Q : RSreceiver | readNextSeq : NZN, fastRepairFlag : X,
                       dataBuffer : MC, repairserver : Q', retransTO : R,
                       dataInfo : (info(NZN', 0, DT, N) DI) > CF }
   =>
   {if (NZN' seqNoIn MC) or (NZN' < NZN) then
    (< Q : RSreceiver | dataInfo : (info(NZN', noTimeValue, DT, N) DI) > CF)
    else (if 48 < N then ERROR
          else (< Q : RSreceiver | dataInfo :
                                     (info(NZN', noTimeValue, R, N) DI) >
                send(NAKPacket(NZN', N, X), Q, Q') CF) fi) fi} .
```

The functions `mte` and `delta` define the "timed" behavior of receiver objects of class `RSreceiverAlone` as follows. The only time-dependent values are the two timers in the information state for each missing data packet. The function `mte` ensures that the tick rule in Section 2.4 stops the time advance when a timer expires, and the function `delta` updates the timers according to the time elapsed:

```
eq mte(< Q : RSreceiverAlone | dataInfo : DI >) = mte(DI) .
op mte : DataInfo -> TimeInf .
eq mte((none).DataInfo) = INF .
ceq mte(DI DI') = min(mte(DI), mte(DI')) if DI =/= none and DI' =/= none .
eq mte(info(NZN, DT, DT', N)) =
      min(if DT =/= noTimeValue then DT else INF fi,
          if DT' =/= noTimeValue then DT' else INF fi) .

eq delta(< Q : RSreceiverAlone | dataInfo : DI >, R) =
      < Q : RSreceiverAlone | dataInfo : delta(DI, R) > .
op delta : DataInfo Time -> DataInfo .
...
```

5 Formal Analysis of the AER/NCA Protocol Suite in Real-Time Maude

This section illustrates how the AER/NCA protocol has been subjected to rapid prototyping and formal analysis. The analysis is described in full detail in [12].

5.1 Rapid Prototyping

To execute the repair service protocol we added a sender application object and a number of receiver application objects, and defined an initial state `RSstate`. The sender was supposed to use the protocol to multicast 21 data packets to the receiver applications. Rewriting this initial state should have led to a state where all receiver applications had received all packets. Instead, the execution gave the following result:

```
Maude> (rew- [3000] RSstate .)

result ClockedSystem : {ERROR} in time 17841
```

By executing fewer rewrites we could follow the execution leading to the `ERROR`-state, and could easily find the errors in the formal and informal specifications. Executing the repair service protocol with a different initial state revealed another undesirable behavior where a lost packet was never repaired, and we could again easily trace the error.

The other protocol components, as well as the composite protocol, have been prototyped by executing initial states, with the desired results:

- Prototyping the RTT protocol resulted in states having the expected values of the round trip times the protocol was supposed to find.
- The NOM protocol was prototyped by placing in the *environment* object (which defines the interface to the other components when a component is executed and analyzed separately) the set of data packets which would be received by the receivers. That way, we knew which object should be the nominee receiver at any time, and executing the protocol indeed produced states having the expected nominees.
- The rate control protocol was prototyped by attempting to send a new data packet every millisecond, and by recording in the state the time stamp of each new data packet sent. The list of sending times could then be inspected to get a feeling for the sending rate, which, as expected, grew (seemingly) exponentially in the beginning.
- The composite protocol was executed with the initial state having the same topology as the one for which execution of the stand-alone RS protocol failed. However, the composite protocol managed to deliver all data packets to each receiver. This was due to the presence of the rate control component, that adjusted the sending rate to avoid the packet losses which led to the faulty behavior in the execution of the RS protocol component.

5.2 Formal Analysis

To substantially increase our confidence in the specifications before costly attempts at formal proofs of correctness, the specifications can be subjected to further formal analysis using the search and model checking commands and the meta-programming features of Real-Time Maude.

For example, the RTT protocol should find in the `sourceRTT` attribute the round trip times from each node to the sender. Likewise, each receiver or repair server should have a `maxUpRTT` value equal to the maximal round trip time from any of its "siblings" to its immediate upstream node. As already mentioned, executing some initial states using Real-Time Maude's default interpreter indeed resulted in states with the expected values of these attributes. To gain further assurance about the correctness of the specification, we have explored not just *one* behavior, arbitrarily chosen by Real-Time Maude's default interpreter, but *all* possible behaviors—relative to certain conditions—starting from the initial state. The main property the stand-alone RTT protocol should satisfy is that, as long as at most one packet travels in the same direction in the same link at the same time, the following properties hold:

- each rewrite path will reach a state with the desired `sourceRTT` and `maxUpRTT` values within given time and depth limits (reachability); and
- once these desired values have been found, they will not change within the given time limit (stability).

We defined an initial test configuration `RTTstate` with nodes `'a`, `'b`, ..., `'g`, and where, in otherwise empty links, the round trip times to the source from the nodes `'c`, `'d`, and `'e` are, respectively, 58, 106, and 94, and the `maxUpRTT` values of these nodes are, respectively, 58, 48, and 48. In a module `varRTT` which extends the specification of the RTT protocol with the declaration of the variables `ATTS1`, `ATTS2`, and `ATTS3` of the sort `AttributeSet` of sets of object attributes, the following pattern is matched by all states where the nodes `'c`, `'d`, and `'e` have the above `sourceRTT` and `maxUpRTT` values:

```
{< 'c : RTTrepairserverAlone | sourceRTT : 58, maxUpRTT : 58, ATTS1 >
 < 'd : RTTrepairserverAlone | sourceRTT : 106, maxUpRTT : 48, ATTS2 >
 < 'e : RTTreceiverAlone | sourceRTT : 94, maxUpRTT : 48, ATTS3 >  CF}.
```

The desired property that the RTT protocol should satisfy can therefore be given by the following temporal formula, where P abbreviates the above pattern:

$$\texttt{anyPattern}\ \mathbf{UStable}_{\leq timeLimit}\ P.$$

Although this property can be model checked by giving a Real-Time Maude command, the tool executes the command too slowly, because it is too general and performs tests which are not necessary in our specification. Instead, we can reuse Real-Time Maude's strategy library to easily define a model checking function `ustable` in a module extending the module `TIMED-META-LEVEL`, where

$$\texttt{ustable}(mod, t_0, n, timeLimit, pattern)$$

gives the set of terms representing rewrite paths using the module mod, starting from the initial term t_0, which *invalidate* the reachability-and-stability property `anyPattern` $\mathbf{UStable}_{\leq timeLimit}$ $pattern$, and which have maximal bound n on the number of rewrites in the path (with 0 meaning unbounded). To further enhance efficiency, `ustable` does not return *all* "bad" states, but only the first state(s) found which invalidate the property. The search returns `emptyTermSet` if the property holds for all paths satisfying the given length bound.

Using Full Maude's `up` function to get the meta-representation of a term, we can check whether the above desired property holds in all rewrite paths having total time elapse less than or equal to 400, starting from state `RTTstate`:

```
Maude> (red ustable(varRTT, up(varRTT, RTTstate), 0, {'400}'Nat,
 up(varRTT,
    {< 'c : RTTrepairserverAlone | sourceRTT : 58, maxUpRTT : 58, ATTS1 >
     < 'd : RTTrepairserverAlone | sourceRTT : 106, maxUpRTT : 48, ATTS2 >
     < 'e : RTTreceiverAlone | sourceRTT : 94, maxUpRTT : 48, ATTS3 > CF})) .)
...
result TermSet : emptyTermSet
```

No path *not* satisfying the desired property was found, increasing our confidence in the correctness of the protocol. To gain further assurance, we could analyze executions starting with other states.

The search function `ustable` has also been used to show the undesired property that there is a behavior—after some receiver has been nominated and is aware of it—in which no receiver has its `isNominee` flag set to `true`. This property can be shown by finding a counterexample to the opposite claim, namely `anyPattern` $\mathbf{UStable}_{\leq\infty}\ P'$, where P' is a pattern stating that some receiver has its `isNominee` flag set to true. In the module `varNOM`, which extends the NOM protocol with a declaration of a variable `ATTS1` of sort `AttributeSet`, the pattern `{< Q : NOMreceiverAlone | isNominee : true, ATTS1 > CF}` is the desired pattern P', which is matched by receivers whose nominee flag is set to true. The property `anyPattern` $\mathbf{UStable}_{\leq\infty}\ P'$ does not hold, and is refuted by providing a counterexample:

```
Maude> (down varNOM : red ustable(varNOM, up(varNOM, NOMstate), 0, noTerm,
  up(varNOM, {< Q : NOMreceiverAlone | isNominee : true, ATTS1 > CF})) .)

result ClockedSystem :
 {< 'e : NOMreceiverAlone | isNominee : false, ... >
  < 'a : NOMsenderAlone | csmNominee : 'e, ... >
  < 'b : NOMreceiverAlone | isNominee : false, ... >
  < 'f : NOMreceiverAlone | isNominee : false, ... >
  < 'g : NOMreceiverAlone | isNominee : false, ... >
  (NAMPacket(true) from 'a to 'e)   ... } in time 19504
```

The NOM protocol has been subjected to further analysis where we have model checked the liveness property that a nominee receiver is always found within a certain amount of time.

6 Conclusions

The work presented in this paper has tested the Real-Time Maude tool and Maude formal methodology with a challenging distributed real-time application, uncovering subtle and important errors in the informal specification. Two key issues for adequate formalization and analysis are the appropriateness and usefulness of the resulting specification, and the adequacy of the tool support. In particular, the formalization needs to be at the right level of abstraction to represent the essential features—including in this case resource contention and real-time behavior—without being overwhelmed by the complex nature of the system being modeled. In this regard, the modularity and composability of the specifications for each component made it easy to understand and analyze individual component and aggregate system behaviors. Furthermore, the flexibility and extensibility of the Real-Time Maude strategy library made it easy to carry out complex analyses tailored to the specific application that would have been infeasible using general purpose algorithms.

There are a number of interesting directions for future work. One is further extensions and optimizations of the Real-Time Maude tool. Another direction involves

developing module calculi suitable for composition of protocol components, providing additional composition mechanisms beyond multiple inheritance. A third research direction is providing additional analytical capabilities, including abstraction transformations that can map some problems into decidable problems, and proof techniques for reasoning about an even richer class of properties.

Acknowledgments. The authors would like to thank S. Bhattacharyya for his help in the initial stages of designing the Maude specification, and Narciso Martí-Oliet for his comments on earlier versions of this paper. This work has been supported by DARPA through Rome Labs. Contract F30602-97-C-0312, by ONR Contract N00014-99-C-0198, and by NSF grants CCR-9900334 and CCR-9900326.

References

1. Active error recovery (AER): AER/NCA software release version 1.1. `http://www.tascnets.com/panama/AER/`, May 2000.
2. E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency - Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*. Springer, 1994.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic*. Computer Science Laboratory, SRI International, Menlo Park, 1999. `http://maude.csl.sri.com`.
4. G. Denker, J. Meseguer, and C. L. Talcott. Formal specification and analysis of active networks and communication protocols: The Maude experience. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*. IEEE, 2000.
5. F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, University of Málaga, 1999.
6. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
7. G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
8. S. Kasera, S. Bhattacharyya, M. Keaton, D. Kiwior, J. Kurose, D. Towsley, and S. Zabele. Scalable fair reliable multicast using active services. Technical Report TR 99-44, University of Massachusetts, Amherst, CMPSCI, 1999.
9. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997. See also UPPAAL home-page at `http://www.uppaal.com/`.
10. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
11. J. Meseguer. Rewriting logic and Maude: a wide-spectrum semantic framework for object-based distributed systems. In S. Smith and C.L. Talcott, editors, *Formal Methods for Open Object-based Distributed Systems, FMOODS 2000*, pages 89–117. Kluwer, 2000.
12. P. C. Ölveczky. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic*. PhD thesis, University of Bergen, 2000. Available at `http://maude.csl.sri.com/papers`.

13. P. C. Ölveczky. Specifying and analyzing the AER/NCA active network protocols in Real-Time Maude. `http://www.csl.sri.com/~peter/AER/AER.html`, 2000.
14. P. C. Ölveczky and J. Meseguer. Real-Time Maude: A tool for simulating and analyzing real-time and hybrid systems. In *Third International Workshop on Rewriting Logic and its Applications*, 2000. To appear in *Electronic Notes in Theoretical Computer Science*.
15. P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. To appear in *Theoretical Computer Science*. Available at `http://maude.csl.sri.com/papers`, September 2000.
16. S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *Automated Deduction–CADE-11*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, 1992.
17. L. C. Paulson. *Isabelle*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.
18. S. Yovine. Kronos: A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1/2), 1997. See also Kronos home-page at `http://www-verimag.imag.fr/TEMPORISE/kronos/`.

Author Index

Astesiano, Egidio 171
Autexier, Serge 269

Bordleau, Francis 156
Browne, James C. 318
Brune, Georg 233

Cerioli, Maura 171
Choppy, Christine 124
Clark, Tony 17
Corriveau, Jean-Pierre 156

Ehrich, Hans-Dieter 32
Erochok, Andrei 249
Eshuis, Rik 76
Evans, Andy 17

Fischer, Clemens 91

Goerigk, Werner 249

Hagerer, Andreas 233, 249
Hammelmann, Bernhard 249
Heckel, Reiko 109
Hennicker, Rolf 300
Hoffman, Piotr 253
Horwitz, Susan 217
Hutter, Dieter 269

Ide, Hans-Dieter 233
Inverardi, Paola 60

Jacobs, Bart 284
Jürjens, Jan 187

Keaton, Mark 333
Kent, Stuart 17
Klin, Bartek 253
Kolodziejczyk-Strunck, Klaus 249
Kurshan, Robert P. 318

Lämmel, Ralf 201
Loginov, Alexey 217

Margaria, Tiziana 233
Meseguer, José 333
Mossakowski, Till 253, 269

Nagelmann, Markus 249
Niese, Oliver 233, 249

Ölveczky, Peter C. 333
Olderog, Ernst-Rüdiger 91
Ostroff, Jonathan S. 2

Paige, Richard F. 2
Pinger, Ralf 32
Poizat, Pascal 124
Poll, Erik 284

Reed, Joy N. 45
Reggio, Gianna 171
Reps, Thomas 217
Reus, Bernhard 300
Royer, Jean-Claude 124

Sauer, Stefan 109
Schröder, Lutz 253
Selic, Bran 1
Sharygina, Natasha 318
Sinclair, Jane E. 45
Steffen, Bernhard 233
Stevens, Perdita 140

Talcott, Carolyn 333
Tarlecki, Andrzej 253

Uchitel, Sebastian 60

Wehrheim, Heike 91
Wieringa, Roel 76
Wirsing, Martin 300

Yong, Suan Hsi 217

Zabele, Steve 333